Compiler-assisted checkpointing of message-passing applications in heterogeneous environments

Gabriel Rodríguez Álvarez

Department of Electronics and Systems
University of A Coruña, Spain

Department of Electronics and Systems
University of A Coruña, Spain

PhD Thesis

Compiler-assisted checkpointing of message-passing applications in heterogeneous environments

Gabriel Rodríguez Álvarez

Diciembre de 2008

PhD Advisors:
María J. Martín Santamaría
and Patricia González Gómez

Dra. María J. Martín Santamaría
Profesora Titular de Universidad
Dpto. de Electrónica e Sistemas
Universidade da Coruña

Dra. Patricia González Gómez
Profesora Titular de Universidad
Dpto. de Electrónica e Sistemas
Universidade da Coruña

CERTIFICAN

Que la memoria titulada "*Compiler-assisted checkpointing of message-passing applications in heterogeneous environments*" ha sido realizada por D. Gabriel Rodríguez Álvarez bajo nuestra dirección en el Departamento de Electrónica e Sistemas de la Universidade da Coruña y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Informática.

A Coruña, 15 de Diciembre de 2008

Fdo.: María J. Martín Santamaría
Codirectora de la Tesis Doctoral

Fdo.: Patricia González Gómez
Codirectora de la Tesis Doctoral

Fdo.: Luís Castedo Ribas
Director del Dpto. de Electrónica e Sistemas

Resumen de la tesis

Introducción

La evolución de la computación paralela hacia infrastructuras *cluster* y *Grid* ha creado nuevas necesidades de tolerancia a fallos. A medida que las máquinas paralelas incrementan su número de procesadores, también se incrementa la tasa de fallos del sistema global. Esto no supone un problema mientras el tiempo medio que una aplicación tarda en terminar su ejecución permanezca por debajo del tiempo medio hasta fallo ($MTTF$) del *hardware*, pero esto no siempre se cumple para aplicaciones con ejecuciones largas. En estas circunstancias, los usuarios y programadores necesitan disponer de mecanismos que les permitan garantizar que no toda la computación realizada se pierde al ocurrir un fallo.

El *checkpointing* se ha convertido en una técnica ampliamente utilizada para la obtención de tolerancia a fallos. Consiste en el almacenamiento periódico del estado de la computación, de forma que la ejecución pueda ser restaurada a partir de dicho estado en caso de fallo. Se han propuesto diversas soluciones y técnicas [22] para lograr este propósito, cada una con sus propias ventajas e inconvenientes.

Las técnicas de *checkpointing* aparecieron originalmente como servicios proporcionados por el sistema operativo (SO), centradas en la recuperación de aplicaciones secuenciales. Ejemplos de este tipo de herramientas son KeyKOS [36], que realizaba *checkpointing* a nivel del sistema, almacenando el estado del operativo completo, y Sentry [56], una implementación basada en *UNIX* que realizaba *checkpointing* y registro de eventos no deterministas para aplicaciones secuenciales. Sprite [45] se centraba en la gestión de la migración de procesos para el *checkpoint* y reinicio de procesos en ejecución en computadores de memoria compartida. Su objetivo era balancear la carga de una red de estaciones de trabajo a través de la migración de procesos que ejecutaban una única aplicación de memoria compartida a máquinas

con baja carga. Al estar implementados en el operativo, estas herramientas eran completamente *ad-hoc*, con falta de énfasis en la portabilidad. Su aproximación a la eficiencia consistía en obtener un buen rendimiento de E/S para el almacenamiento del estado de la computación, en lugar de reducir la cantidad de datos a almacenar.

La primera desventaja obvia de las implementaciones basadas en el operativo es la dependencia existente entre la tolerancia a fallos y el sistema operativo concreto en uso. Las herramientas de *checkpointing* comunes en los operativos hasta los 90, comenzaron a desparecer en entornos populares como *UNIX*, *SunOS* o *AIX*. La necesidad de soluciones flexibles capaces de operar en entornos diferentes motivaron la aparición de las soluciones a nivel de aplicación (opuestas a las soluciones a nivel de sistema). En estas herramientas, la tolerancia a fallos se consigue a través de la compilación de la aplicación junto con el código de *checkpointing* ubicado habitualmente en una librería aparte. Las herramientas de *checkpointing* en este período eran aún transparentes, almacenando el estado de la aplicación al completo. Al no implementarse dentro del *kernel* del operativo debían enfrentarse a problemas importantes al manipular estado dependiente del SO. Ejemplos son la recuperación de los identificadores de procesos o de la tabla de ficheros abiertos. Además, debían disponer de mecanismos de recuperación de la pila o el montículo. Estos problemas hacían que el código de las herramientas fuese aún muy dependiente de los servicios proporcionados por el operativo. Normalmente, esto forzaba a los desarrolladores a restringir el tipo de servicios que una aplicación podía usar. Ejemplos de herramientas transparentes a nivel de aplicación son Libckpt [49], CATCH GCC [38] (una versión modificada del compilador GNU C), y la herramienta uniprocesador incluída con el sistema Condor [41].

A mitad de los 90 comenzaron a aparecer soluciones no transparentes que intentaban aplicar *checkpointing* a plataformas de computación distribuida. Su desventaja fundamental era la falta de una interfaz estándar para la comunicación entre procesos. Ejemplos de estas herramientas son Calypso [8] y las extensiones a Dome (*Distributed Object Migration Enviroment*) [10]. En ambos casos el lenguaje de programación objetivo era una extensión de C++ con construcciones paralelas no estándar.

La adopción de MPI como el estándar *de-facto* para programación paralela ha motivado la aparición de herramientas de *checkpointing* basadas en esta interfaz en los últimos años. Al principio, estas herramientas usaban la aproximación transparente a nivel de aplicación, compartiendo las desventajas de sus primas para aplicaciones secuenciales: falta de portabilidad de los datos y restricción de los entornos

soportados, que aquí se refiere a la implementación concreta de MPI. De hecho, los *checkpointers* en esta categoría se implementan generalmente mediante la modificación de una librería MPI previamente existente. Ejemplos de este tipo de *checkpointers* son MPICH-GF [71] y MPICH-V2 [12], ambas implementadas como drivers MPICH, obligando por tanto a todas las máquinas a disponer de esta implementación concreta de MPI.

Las nuevas tendencias de computación, tales como el uso de grandes *clusters* heterogéneos y sistemas *Grid*, presentan nuevas restricciones para las técnicas de *checkpointing*. La heterogeneidad hace imposible aplicar soluciones tradicionales de almacenamiento del estado de la computación, que usan estrategias no portables para la recuperación de estructuras como la pila, el montículo, o el estado de las comunicaciones de las aplicaciones.

Así, las nuevas técnicas de *checkpointing* deben proporcionar estrategias para la recuperación portable del estado de la ejecución, donde ésta pueda ser restaurada en un amplio rango de máquinas, ya sean éstas incompatibles a nivel binario o usen versiones incompatibles de utilidades *software*, tales como interfaces de comunicaciones entre procesos.

Este trabajo presenta CPPC (*ComPiler for Portable Checkpointing*), una herramienta de *checkpointing* centrada en la inserción automática de tolerancia a fallos en aplicaciones de paso de mensajes. Ha sido diseñada para permitir el reinicio de las ejecuciones en diferentes arquitecturas y/o sistemas operativos, permitiendo también el *checkpointing* en sistemas heterogéneos. Utiliza código y protocolos portables, y genera ficheros de estado portables al mismo tiempo que evita el uso de soluciones tradicionales, como la coordinación de procesos o el registro de mensajes (*message-logging*) que están ligadas a falta de escalabilidad.

Metodología de trabajo

El primer paso en la elaboración de la tesis ha sido la revisión de la bibliografía existente en el ámbito de la tolerancia a fallos de aplicaciones paralelas. Establecidas las ventajas y general predominancia de las soluciones basadas en *checkpointing*, se ha abordado el diseño partiendo de aplicaciones extremadamente sencillas, construyendo soluciones ad-hoc y continuando con ejemplos cada vez más complejos y tratando de abstraer lo común a todas ellas. A partir de estas experiencias, se ha construido una librería que proporcione apoyo a las tareas comunes necesarias, que

fue posteriormente probada mediante utilización manual para la inserción de tolerancia a fallos en diversas aplicaciones científicas de complejidad superior a las ya estudiadas. En caso de detectar alguna falla, esta fase ha servido para mejorar la librería, realizando las mejoras necesarias que podían preverse en la etapa de análisis inicial.

Una vez disponible la librería funcional utilizada para la obtención manual de tolerancia a fallos, se ha procedido a trabajar en la construcción de la herramienta de transformación, haciendo uso del conocimiento adquirido mediante la transformación manual de las aplicaciones, generalizando las técnicas utilizadas y realizando implementaciones automáticas de dichas transformaciones.

En todo momento, se ha procurado, con vistas a cumplir el objetivo de usabilidad en entornos heterogéneos, que las soluciones implementadas tengan un diseño tal que permita su utilización en diferentes entornos de trabajo, bien mediante técnicas de ingeniería de software específicas para dicho fin o mediante el uso de herramientas de programación que permitan tal característica de forma nativa.

Una vez implementado el grueso de la infraestructura, se ha procedido a realizar una metódica evaluación experimental de su comportamiento.

Contribuciones

Este trabajo presenta técnicas novedosas para el *checkpointing* de aplicaciones paralelas, dirigidas a la obtención de las siguientes propiedades relevantes de tolerancia a fallos:

1. Independencia del SO: las estrategias de *checkpointing* deben ser compatibles con un amplio espectro de sistemas operativos. Esto implica el disponer de, al menos, una estructura modular básica que permita la sustitución de ciertas secciones críticas de código (p. ej. acceso al sistema de ficheros) dependiendo del SO subyacente.

2. Soporte de aplicaciones paralelas con independencia del protocolo de comunicaciones: la herramienta de *checkpointing* no debería presuponer el uso de una interfaz de comunicaciones determinada. Los *Grid*, por ejemplo, incluyen máquinas pertenecientes a diferentes entidades independientes que no pueden ser obligadas a incluir una versión determinada de la interfaz MPI. Incluso

reconociendo el rol de MPI como estándar *de-facto* para el paso de mensajes, la técnica de *checkpointing* no puede estar ligada a nivel teórico con MPI si debe proporcionar una solución portable y reutilizable.

3. Reducción del tamaño de los ficheros de estado: la herramienta debe optimizar la cantidad de datos almacenados, evitando incluir datos no necesarios para el reinicio de la aplicación. Esto mejora el rendimiento, que depende en gran medida del tamaño de los ficheros. También mejora el rendimiento de la migración de los procesos, en caso de ser necesaria.

4. Recuperación de datos portable: el estado de la aplicación debe ser visto como una estructura que contiene diversos tipos de datos. La herramienta de *checkpointing* debe ser capaz de recuperar todos estos datos de forma portable. Esto incluye la recuperación de estado opaco, como comunicadores MPI, así como de estado dependiente del operativo, como la tabla de ficheros abiertos o la pila.

Con respecto a la independencia del SO, se plantea la construcción de una herramienta con una elevada carga de modularidad, que posibilite el reemplazo dinámico de la implementación de diversas utilidades como puede ser el acceso al sistema de ficheros o las comunicaciones entre procesos. En este aspecto cobra importancia el protocolo de reinicio propuesto, basado en la re-ejecución selectiva de código para la recuperación de las partes críticas del estado que no pueden ser simplemente almacenadas mediante el uso de formatos portables. Para la portabilidad del compilador, se emplean técnicas basadas en el suministro de datos referentes a las semánticas de las diversas funciones utilizadas por la aplicación a instrumentar (catálogos semánticos). El suministro de estos datos se realiza mediante ficheros XML que documentan librerías de funciones ampliamente utilizadas (p. ej. funciones *POSIX*) proporcionados junto con la herramienta, de forma que el usuario no sea responsable de su creación.

El soporte de aplicaciones paralelas con independencia del protocolo de comunicaciones utilizado se posibilita, además de por el uso de los catálogos semánticos anteriormente mencionados, mediante el uso de la técnica de coordinación estática de los procesos en tiempo de compilación. La creación de los ficheros de estado en puntos seguros proporciona, además, un alto nivel de escalabilidad a la herramienta. El compilador incluye análisis de flujo de comunicaciones y de complejidad de código que posibilitan la automatización del proceso de inserción de *checkpoints* en dichos puntos seguros.

La reducción del tamaño de los ficheros, y por tanto la operación eficiente, se consigue mediante el almacenamiento de los datos a nivel de variable, esto es, los datos almacenados en los ficheros de estado son variables de usuario y metadatos que posibilitan la recuperación de éstos durante el proceso de re-ejecución selectiva llevado a cabo en el reinicio de las aplicaciones. El compilador incluye análisis de variables vivas que automatizan la selección de las variables de usuario a almacenar.

Para la recuperación portable de todos los datos de la aplicación se introduce, aparte del protocolo de reinicio mediante re-ejecución selectiva, un formato jerárquico de los ficheros que permite la recuperación de los datos en el ámbito de ejecución adecuado. El formato permite también almacenar los determinantes de los eventos no deterministas (en este caso modelados como la ejecución de rutinas no portables) de forma que dichos eventos puedan ser re-ejecutados durante el reinicio en idénticas condiciones, restaurando así el estado de forma semánticamente apropiada y con independencia de la implementación concreta de las rutinas no portables objeto de la re-ejecución.

Este trabajo presenta también una evaluación experimental de la herramienta novedosa, dado que se ha realizado en un entorno de computación de altas prestaciones público (el Finis Terrae del Centro de Supercomputación de Galicia), a diferencia de muchas otras herramientas que realizan su evaluación en entornos cerrados y altamente controlados. Para ello, ha sido necesaria la realización de un elevado volumen de experimentos junto con el uso de técnicas de análisis estadístico de los datos arrojados por los mismos. Las pruebas se han realizado sobre un gran número de códigos públicamente disponibles, y se ha proporcionado también gran cantidad de información sobre la instrumentación introducida en cada código, de forma que sea posible la repetición y el contraste de los experimentos.

Conclusiones

CPPC es una herramienta de *checkpointing* transparente para aplicaciones de paso de mensajes con ejecuciones largas. Está formada por una librería que contiene rutinas de *checkpointing*, junto con un compilador que automatiza el uso de la librería. Las aspectos más destacables de esta infraestructura son:

- El protocolo de coordinación en tiempo de compilación: la consistencia global de los diferentes ficheros de estado locales se consigue a través de la ubicación

en puntos seguros de los *checkpoints*. De este modo, las operaciones relacionadas con la consistencia se transfieren de la ejecución normal de las aplicaciones a la compilación y el reinicio de las mismas. Durante la compilación, se detectan los puntos seguros y se seleccionan algunos de ellos para la introducción de *checkpoints*. Durante el reinicio los procesos involucrados en la ejecución de la aplicación se comunican entre sí y deciden desde cuál de los *checkpoints* debe realizarse el dicho reinicio. La sincronización de procesos utilizada por las aproximaciones tradicionales se realiza durante el reinicio, mejorando así la escalabilidad de la solución. Además, se mejora la eficiencia, dado que tanto la compilación como el reinicio son operaciones mucho menos frecuentes que la creación de los ficheros de estado.

- La reducción del tamaño de los ficheros de estado: CPPC utiliza una aproximación a nivel de variable, almacenando sólo aquellas variables que son necesarias durante el reinicio. Al restringir la cantidad de datos almacenados, se disminuye el tamaño de los ficheros de estado y se reduce la sobrecarga del *checkpointing*. Esto mejora también el rendimiento de las transferencias de los mismos a través de la red, si fuera necesario.

- La recuperación portable del estado de la aplicación: la recuperación del estado se realiza de forma portable mediante el formato jerárquico utilizado para el volcado de los datos y el proceso de re-ejecución selectiva de código no-portable que tiene lugar durante el reinicio. Esta re-ejecución proporciona también mecanismos para la recuperación del estado de librerías externas en uso por la aplicación. La portabilidad es una propiedad interesante debido a la inherente heterogeneidad de las arquitecturas en boga para la computación de altas prestaciones, como el *Grid*. CPPC-G [55] es un proyecto en desarrollo basado en la creación de una arquitectura de servicios web para dar soporte a la ejecución de aplicaciones CPPC en entornos *Grid*.

- El diseño modular de la librería: la librería CPPC está implementada en C++, utilizando un diseño altamente modular que permite la configuración flexible de todas sus funcionalidades. Permite configurar dinámicamente el formato de los ficheros de estado, así como activar varias características opcionales como la compresión de los mismos, importante cuando se trabaja sobre redes lentas o con espacio en disco limitado, o comprabaciones de integridad. El diseño modular permite también la utilización de la herramienta en entornos donde no se dispone de derechos de administración, lo que es común al

trabajar con computadores de altas prestaciones. El uso del patrón modelo-vista-controlador permite las llamadas a la librería desde diversos lenguajes de programación, mediante la implementación de una delgada capa de *software* que adapte las peticiones de la aplicación a la interfaz interna del núcleo CPPC.

- El *checkpointing* completamente transparente: el compilador CPPC convierte a la librería CPPC en una herramienta de *checkpointing* totalmente transparente, pues automatiza todos los análisis y transformaciones necesarios para el uso de la misma. El compilador incluye también una novedosa técnica para la identificación automática de lazos intensivos en computación basada en el análisis de métricas de complejidad de código típicas de la ingeniería del *software*.

CPPC ha sido evaluado experimentalmente demostrando su usabilidad, escalabilidad, eficiencia y portabilidad. Es capaz de reiniciar aplicaciones en diferentes arquitecturas y/o máquinas utilizando diferentes compiladores C/Fortran e implementaciones MPI, todo ello usando el mismo conjunto de ficheros de estado. El rendimiento de la herramienta se ha estimado usando aproximaciones estadísticas sobre los resultados obtenidos de ejecuciones en una infraestructura de computación pública (el supercomputador Finis Terrae del CESGA).

Hasta donde alcanza nuestro conocimiento, CPPC es el único *checkpointer* de aplicaciones de paso de mensajes disponible en el dominio público. CPPC es un proyecto de código abierto, disponible en `http://cppc.des.udc.es` bajo licencia GPL.

El desarrollo de esta tesis ha dado lugar a las siguientes publicaciones:

- G. Rodríguez, M.J. Martín, P. González, J. Touriño y R. Doallo. Controlador/preCompilador de Checkpoints Portables. En *Actas de las XV Jornadas de Paralelismo*, pp. 253–258, Almería, Septiembre de 2004.
- G. Rodríguez, M.J. Martín, P. González, J. Touriño y R. Doallo. On designing portable checkpointing tools for large-scale parallel applications. En *Proceedings of the 2nd International Conference on Computational Science and Engineering (ICCSE'05)*, pp. 191–203. Estambul (Turquía), Junio de 2005.
- G. Rodríguez, M.J. Martín, P. González, J. Touriño y R. Doallo. Portable Checkpointing of MPI applications. En *Proceedings of the 12th Workshop on*

Compilers for Parallel Computers (CPC'06), pp. 396–410, A Coruña, Enero de 2006.

- G. Rodríguez, M.J. Martín, P. González y J. Touriño. Controller/Precompiler for Portable Checkpointing. En *IEICE Transactions on Information and Systems*, Vol. E89-D, No. 2, pp. 408–417, Febrero de 2006.

- G. Rodríguez, M.J. Martín, P. González, J. Touriño y R. Doallo. CPPC: Una herramienta portable para el checkpointing de aplicaciones paralelas. en *Boletín de la red nacional de I+D, RedIRIS*, No. 80, pp. 57–61, Abril de 2007.

- D. Díaz, X.C. Pardo, M.J. Martín, P. González y G. Rodríguez. CPPC-G: Fault-Tolerant Parallel Applications on the Grid. En *Proceedings of the 1st Iberian Grid Infrastructure Conference (IBERGRID'07)*, pp. 230–241, Santiago de Compostela, Mayo de 2007.

- G. Rodríguez, P. González, M.J. Martín y J. Touriño. Enhancing Fault-Tolerance of Large-Scale MPI Scientific Applications. En *Proceedings of the 9th International Conference on Parallel Computing Technologies (PaCT'07)*, volumen 4671 de *Lecture Notes in Computer Science*, pp. 153–161, Pereslavl-Zalessky (Rusia), Septiembre de 2007.

- D. Díaz, X.C. Pardo, M.J. Martín, P. González y G. Rodríguez. CPPC-G: Fault Tolerant Parallel Applications on the Grid. En 3rd Workshop on Large Scale Computations on Grids (LaSCoG'07). *Lecture Notes in Computer Science*, volumen 4967, pp. 852–859, Mayo de 2008. ISBN 978-3-540-68105-2.

- G. Rodríguez, X.C. Pardo, M.J. Martín, P. González, D. Díaz. A Fault Tolerance Solution for Sequential and MPI Applications on the Grid. En *Scalable Computing: Practice and Experience*, Vol. 9, No. 2, pp. 101–109, Junio de 2008. ISSN 1895-1767.

- G. Rodríguez, M.J. Martín, P. González, J. Touriño, R. Doallo. CPPC: A Compiler-Assisted Tool for Portable Checkpointing of Message-Passing Applications. En *Proceedings of the 1st International Workshop on Scalable Tools for High-End Computing (STHEC'08), held in conjunction with the 22nd ACM International Conference on Supercomputing (ICS'08)*, pp. 1–12, Kos (Grecia), Junio de 2008.

Trabajo Futuro

Hay dos aspectos fundamentales en los que mejorar la herramienta CPPC. En primer lugar, la coordinación espacial de procesos tiene una desventaja importante: la necesidad de especificar la frecuencia de *checkpointing* como un parámetro dependiente del número de llamadas a la función de *checkpointing*, en lugar de en función del tiempo. Actualmente estamos trabajando en un protocolo ligero y no coordinado de comunicaciones que permite variar dinámicamente la frecuencia espacial en función de una frecuencia temporal especificada por el usuario. Este protocolo se basa en la comunicación no bloqueante entre todos los procesos de información relativa a la velocidad a la que cada uno de ellos ejecuta el código para, posteriormente, adoptar una frecuencia espacial que garantice que el proceso más lento crea ficheros de estado con una frecuencia similar a la especificada.

La segunda mejora está relacionada con los análisis de complejidad llevados a cabo para la inserción de *checkpoints* en el código. El algoritmo actualmente usado en el compilador CPPC puede ser mejorado a través del uso de métricas más complejas, de forma que sea posible obtener resultados más próximos a los óptimos.

Abstract

With the evolution of high performance computing towards heterogeneous, massively parallel systems, parallel applications have developed new checkpoint and restart necessities. Whether due to a failure in the execution or to a migration of the processes to different machines, checkpointing tools must be able to operate in heterogeneous environments. However, some of the data manipulated by a parallel application are not truly portable. Examples of these include opaque state (e.g. data structures for communications support) or diversity of interfaces for a single feature (e.g. communications, I/O). Directly manipulating the underlying ad-hoc representations renders checkpointing tools incapable of working on different environments. Portable checkpointers usually work around portability issues at the cost of transparency: the user must provide information such as what data needs to be stored, where to store it, or where to checkpoint. CPPC (ComPiler for Portable Checkpointing) is a checkpointing tool designed to feature both portability and transparency, while preserving the scalability of the executed applications. It is made up of a library and a compiler. The CPPC library contains routines for variable level checkpointing, using portable code and protocols. The CPPC compiler achieves transparency by relieving the user from time-consuming tasks, such as performing code analyses and adding instrumentation code.

Index terms — Fault tolerance, checkpointing, parallel programming, message-passing, MPI, compiler support.

Acknowledgements

Many people have contributed to the work presented in this thesis. My Ph.D. advisors, María and Patricia, have made it possible with their constant support and hard work, as well as their sheer patience. Although not an advisor, Juan has also greatly contributed to the development of this work through both his efforts and his invaluable sense of criticism. I also want to extend my thanks to the people at the Computer Architecture Group, particularly to its head Ramón Doallo, and at the Department of Electronics and Systems.

On a personal level, this work would not have been the same without the solid anchorage point my family provides. And certanly not the same without María, who should be thanked for all the things that do not fit on this page. Also on a personal level – albeit admittedly a geeky one – Marcos and José have a habit of making things always brighter and sharper through their ideas, criticism, and also their moaning and whining (strictly when necessary).

I also want to acknowledge the following persons and institutions: CESGA (Galician Supercomputing Center) for providing access to their computing resources, particularly the Finis Terrae supercomputer, and specially to José Carlos Mouriño for helping me out in my struggles with their computing environment; the Laboratory for Advanced Systems Research of the University of Texas at Austin, specially to Prof. Lorenzo Alvisi, for hosting me during my research visit in 2007; Dr. Volker Strumpen of IBM and Prof. Keshav Pingali of the University of Texas at Austin for their suggestions for the improvement of this work; and the VARIDIS research group of the Politechnical University of Catalonia for providing access to their Fekete application.

I gratefully thank the following institutions for funding this work: the Department of Electronics and Systems of A Coruña for the human and material support; the University of A Coruña for financing my attendance to some conferences; the

Xunta de Galicia (project PGIDIT04TIC105004PR); the Ministry of Science and Innovation of Spain (FPU grant AP-2004-2685 and projects TIN2004-07797-C02-02 and TIN2007-67537-C03-02); and CYTED (Iberoamerican Programme of Science and Technology for Development; project 506PI0293).

"Research is what I'm doing when I don't know what I'm doing"

– *Wernher Von Braun*

Contents

Preface **1**

1. Fault tolerance for parallel applications **7**

1.1. Approaches for parallel fault tolerance 8

1.1.1. Using intercommunicators 8

1.1.2. Modifications to MPI semantics 9

1.1.3. MPI extensions 9

1.1.4. Rollback-recovery 10

1.2. Checkpoint-based approaches 11

1.2.1. Uncoordinated checkpointing 11

1.2.2. Coordinated checkpointing 13

1.2.3. Communication-induced checkpointing 15

1.3. Log-based approaches 15

1.3.1. Pessimistic log-based approaches 16

1.3.2. Optimistic log-based approaches 17

1.3.3. Causal log-based approaches 18

1.4. Implementation properties of rollback-recovery protocols 18

1.4.1. Granularity 18
1.4.2. Transparency 20
1.4.3. Portability 20
1.5. Existing checkpointing tools 21
1.5.1. CoCheck 21
1.5.2. CLIP 21
1.5.3. Porch 22
1.5.4. Egida 22
1.5.5. Starfish 23
1.5.6. MPICH-V2 23
1.5.7. MPICH-GF 24
1.5.8. PC^3 24
1.6. Proposal 24
1.6.1. Spatial coordination 25
1.6.2. Granularity and portability 27
1.7. Summary 28

2. CPPC Library **31**

2.1. View 31
2.1.1. CPPC initialization and shutdown 32
2.1.2. Variable registration 34
2.1.3. Non-portable calls 35
2.1.4. Context management 37
2.1.5. Open files 38

2.1.6. Checkpoint file dumping 39
2.1.7. Application restart 39
2.2. Controllers 41
2.3. Model 42
2.3.1. Façade 43
2.3.2. Checkpointing layer 53
2.3.3. Writing layer 56
2.3.4. Portability layer 57
2.3.5. Utility layer 59
2.4. Summary 60

3. CPPC Compiler **63**

3.1. Compiler overview 63
3.2. Analyses and transformations 67
3.2.1. CPPC initialization and finalization routines 67
3.2.2. Procedure calls with non-portable outcome 68
3.2.3. Open files 68
3.2.4. Conversion to CPPC statements 69
3.2.5. Data flow analysis 70
3.2.6. Communication analysis 73
3.2.7. Checkpoint insertion 78
3.2.8. Language-specific transformations 81
3.2.9. Code generation 82
3.3. Case study 82

3.4. Implementation details 86
3.4.1. Fortran 77 support 86
3.4.2. Sharing code between the C and Fortran 77 compilers 88
3.4.3. AST analyzers 89
3.5. Related work 91
3.6. Summary 92

4. Experimental Results **95**
4.1. Introduction 95
4.2. Compiler 98
4.2.1. Analysis of the instrumented codes 98
4.2.2. Compilation times 118
4.3. Library 119
4.3.1. State file sizes 123
4.3.2. State file creation time 124
4.3.3. Checkpoint overhead 131
4.3.4. Restart times 134
4.4. Summary 137

Conclusions and future work **141**

Bibliography **145**

List of Tables

2.1. Interface summary of the CPPC library 33

2.2. CPPC configuration parameters 45

3.1. Semantic roles used by the CPPC compiler 66

4.1. Summary of test applications 96

4.1. Summary of test applications (continued) 97

4.2. Detail of loops selected by the shape-based threshold for NAS BT . . 100

4.3. Detail of loops selected by the shape-based threshold for NAS CG . . 102

4.4. Detail of loops selected by the shape-based threshold for NAS EP . . 102

4.5. Detail of loops selected by the shape-based threshold for NAS FT . . 104

4.6. Detail of loops selected by the shape-based threshold for NAS IS . . . 107

4.7. Detail of loops selected by the shape-based threshold for NAS LU . . 107

4.8. Detail of loops selected by the shape-based threshold for NAS MG . . 109

4.9. Detail of loops selected by the shape-based threshold for NAS SP . . 109

4.10. Detail of loops selected by the shape-based threshold for CESGA CalcuNetw 112

4.11. Detail of loops selected by the shape-based threshold for CESGA Fekete 115

4.12. Detail of loops selected by the shape-based threshold for DBEM . . . 118

4.13. Detail of loops selected by the shape-based threshold for STEM-II . . 118

4.14. Statistics and compilation times for test applications 120

4.15. Breakdown of compilation times for test applications 121

4.16. Number of nodes and cores used for runtime tests for each application 122

4.17. Performance of the automatic variable registration algorithm 124

4.18. Checkpoint file creation times (seconds) 132

4.19. Runtime overhead caused by checkpointing 135

4.20. Runtime overhead on large-scale applications 136

4.21. Restart times (seconds) 137

List of Figures

1. Design of the CPPC framework 4

1.1. Domino effect in uncoordinated checkpointing 13

1.2. Spatial coordination for non-blocking coordinated checkpointing . . . 26

2.1. Global design of the CPPC framework 43

2.2. Pseudocode of the algorithm for finding the recovery line 47

2.3. Data hierarchy format used for writing plugins 52

3.1. Semantic information for the `fopen` and `open` functions 65

3.2. Example of a non-portable procedure call transformation 68

3.3. Pseudocode for a file opening transformation 69

3.4. Pseudocode for a file closing transformation 69

3.5. First step: shape-based threshold 79

3.6. Second step: cluster-based threshold 80

3.7. Modifications to a checkpointed Fortran 77 loop 81

3.8. Case study: original DBEM code 83

3.9. Case study: CPPC-instrumented DBEM code 84

3.10. Case study: CPPC-instrumented DBEM code (cont.) 85

3.11. AST analyzer implementation by instantiation of the method dispatcher 90

4.1. Checkpoint insertion for NAS BT 99

4.2. Checkpoint insertion for NAS CG 101

4.3. Checkpoint insertion for NAS EP 103

4.4. Checkpoint insertion for NAS FT 105

4.5. Checkpoint insertion for NAS IS 106

4.6. Checkpoint insertion for NAS LU 108

4.7. Checkpoint insertion for NAS MG 110

4.8. Checkpoint insertion for NAS SP 111

4.9. Checkpoint insertion for CESGA CalcuNetw 113

4.10. Checkpoint insertion for CESGA Fekete 114

4.11. Checkpoint insertion for DBEM 116

4.12. Checkpoint insertion for STEM-II 117

4.13. File sizes for NAS BT 125

4.14. File sizes for NAS CG 125

4.15. File sizes for NAS EP 126

4.16. File sizes for NAS FT 126

4.17. File sizes for NAS IS 127

4.18. File sizes for NAS LU 127

4.19. File sizes for NAS MG 128

4.20. File sizes for NAS SP 128

4.21. File sizes for CESGA Fekete 129

4.22. File sizes for DBEM 129

4.23. File sizes for STEM-II 130

4.24. Summary of file sizes 130

4.25. Maximum mean dumping times for test applications 133

4.26. Restart times for test applications 138

Preface

Parallel computing evolution towards cluster and Grid infrastructures has created new fault tolerance needs. As parallel machines increase their number of processors, so does the failure rate of the global system. This is not a problem as long as the mean time to complete an application's execution remains well under the mean time to failure (MTTF) of the underlying hardware, but that is not always true for applications with large execution times. Under these circumstances, users and programmers need a way to ensure that not all computation done is lost on machine failures.

Checkpointing has become a widely used technique to obtain fault tolerance. It periodically saves the computation state to stable storage so that the application execution can be resumed by restoring such state. A number of solutions and techniques have been proposed [22], each having its own pros and cons.

Current trends towards new computing infrastructures, such as large heterogeneous clusters and Grid systems, present new constraints for checkpointing techniques. Heterogeneity makes it impossible to apply traditional state saving techniques which use non-portable strategies for recovering structures such as application stack, heap, or communication state.

Therefore, modern checkpointing techniques need to provide strategies for portable state recovery, where the computation can be resumed on a wide range of machines, from binary incompatible architectures to incompatible versions of software facilities, such as different implementations for communication interfaces.

This work presents a checkpointing framework focused on the automatic insertion of fault tolerance into long-running message-passing applications. It is designed to allow for execution restart on different architectures and/or operating systems, also supporting checkpointing over heterogeneous systems, such as the Grid. It uses portable code and protocols, and generates portable checkpoint files while avoiding

traditional solutions (such as process coordination or message logging) which add an unscalable overhead.

Checkpointing evolution

Checkpointing techniques appeared as operating system (OS) services, usually focused on recovery of sequential applications. Examples are KeyKOS [36], which performed system-wide checkpointing, storing the entire OS state, and the Sentry [56], a UNIX-based implementation which performed checkpointing and journaling for logging of non-deterministic events on single processes. Sprite [45] dealt with the process migration aspect of checkpoint-and-restart recovery for shared memory computers. Its focus was balancing the workload of a network of workstations by migrating all the processes executing a single shared-memory application to idle machines. Being implemented into the OS, these checkpointing solutions were completely ad-hoc, with a lack of emphasis on portability. Their approach to operation efficiency was based on achieving good I/O performance for storing the whole computation state, instead of reducing the amount of data to be stored.

The first obvious disadvantage of OS-based implementations is the hard dependency between fault tolerance and the operating system of choice. Checkpointing facilities, which were a common feature in earlier operating systems, gradually disappeared in the early 90s and were unavailable for popular environments such as UNIX, SunOS or AIX. The desire for flexible solutions which could operate in different environments motivated the emergence of application level solutions (as opposed to system level solutions). In these tools, fault tolerance is achieved by compiling the application program together with the checkpointing code, usually found in a library. Checkpointing solutions in this period were still transparent, storing the entire application state. Not being implemented inside the kernel they had to solve important problems when manipulating OS-dependent state. Examples are restoring process identifiers or tracing open files. Also, they had to figure out ways to recover the application stack or heap. These issues made their codes still very dependent on specific operating system features. Usually, this forced developers to restrict the type of OS facilities used by the checkpointed programs. Examples of application level, transparent tools include Libckpt [49], CATCH GCC [38] (a modified version of the GNU C compiler), and the uniprocessor checkpointing facility included with the Condor batch system [41].

Also in the mid-90s some non-transparent solutions tried to apply checkpointing to distributed platforms. Their fundamental drawback was the lack of common ground regarding the interface for interprocess communication, which made these solutions tied to a specific and non-standard interface. Examples of these frameworks are Calypso [8] and extensions to Dome (Distributed Object Migration Environment) [10]. In both cases the programming language used was an extension of C++ with non-standard parallel constructs.

The adoption of MPI as the de-facto standard for parallel programming spawned the appearance of many MPI-based checkpointing tools in the last years. At first, these used the transparent application level approach, sharing the same drawbacks as their uniprocessor counterparts: lack of data portability and restriction of supported environments, which here refers to the underlying MPI implementation. In fact, checkpointers in this category are generally implemented by modifying a previously existing MPI library. Examples of these types of checkpointers are MPICH-GF [71] and MPICH-V2 [12], both implemented as MPICH drivers, thus forcing all machines to run this specific implementation.

More recently, the advent of the Grid requires that checkpointers evolve towards application level approaches that enable both data portability, by storing data using portable representation formats, and communication-layer independence, by implementing the solution in a higher level of abstraction.

The CPPC Framework

As stated in the previous section, modern computing trends require portable tools for message-passing applications, focusing on providing the following fundamental features:

1. OS-independence: checkpointing strategies must be compatible with any given operating system. This means having at least a basic modular structure to allow for substitution of certain critical sections of code (e.g. filesystem access) depending on the underlying OS.

2. Support for parallel applications with communication protocol independence: the checkpointing framework should not make any assumption as to the communication interface or implementation being used. Computational Grids include machines belonging to independent entities which cannot be forced to

Figure 1: Design of the CPPC framework

provide a certain version of the MPI interface. Even recognizing the role of MPI as the message-passing de-facto standard, the checkpointing technique cannot be theoretically tied to MPI in order to provide a truly portable, reusable approach.

3. Reduced checkpoint file sizes: the tool should optimize the amount of data being saved, avoiding dumping state which will not be necessary upon application restart. This improves performance, which depends heavily on state file sizes. It also enhances performance of the process migration in computational Grids.

4. Portable data recovery: the state of an application can be seen as a structure containing different types of data. The checkpointing tool must be able to recover all these data in a portable way. This includes recovery of opaque state, such as MPI communicators, as well as of OS-dependent state, such as the file table or the execution stack.

The CPPC framework (ComPiler for Portable Checkpointing) provides all these features which are key issues for fault-tolerance support on heterogeneous systems. It appears to the user as a runtime library containing checkpoint-support routines [54], together with a compiler which automates the use of the library. The global process is depicted in Figure 1. Upon runtime, the fault tolerant parallel application performs calls to the routines provided by the CPPC Library.

Organization of this Thesis

The structure of this work is as follows. Chapter 1 describes the various alternatives for fault tolerance in parallel applications. It surveys the existing rollback-recovery tools focusing on MPI applications. It also describes the CPPC proposal, outlining its most important characteristics and design decisions.

Chapter 2 provides an in-depth description of the design and implementation of the CPPC library following a top-down approach, starting at the application code

and descending through the various layers in which the library is divided. It also focuses on the modifications that need to be performed to the original code in order to achieve integration with CPPC.

In Chapter 3 the design and implementation of the CPPC source-to-source compiler are covered. It also provides details on the design decisions taken to enable the reusability of the platform. Finally, it includes a related work section covering research focused on the static analysis and checkpoint insertion of programming codes.

Chapter 4 describes the thorough experimental evaluation of the entire CPPC framework, including results to assess the performance of both the compiler and the checkpointing library. The experimental tests were performed on the Finis Terrae supercomputer hosted by the Galician Supercomputing Center (CESGA). This provides a more accurate performance estimation than using highly controlled and closed environments available to a particular research group only.

Finally, the work is concluded by outlining its most relevant aspects and discussing future research lines.

Chapter 1

Fault tolerance for parallel applications

The need for an evergrowing computational power has always been one of the fundamental driving forces of computer evolution, motivated by emerging requirements in such application fields as sciences, engineering, or even leisure. The traditional approach for this evolution has been the enhancement of already-known technologies: building Von Neumann machines with faster processors and larger amounts of memory. This approach is not always feasible, however. There are physical limits imposed by several factors, such as the machines in charge of circuit integration, or the amount of energy the circuits are able to dissipate as heat. There are also economic constraints: the cost of new developments does not achieve return-on-investment for certain projects. These limitations are one of the most important reasons for the popularization of parallel computing: an approach that seeks to break down a problem into subtasks which can be solved in a concurrent fashion.

Parallel applications are a special target for fault tolerance. Interprocess communications generate dependencies between processes participating in a parallel execution. Therefore, fault tolerance mechanisms cannot be applied individually to each process in an independent way. Simply recovering a failed process can break the established dependencies, thereby creating inconsistencies at the application level.

Conceptually different approaches to solving the parallel fault tolerance problem have been developed. Although rollback-recovery is the most widely used, a brief survey of other options is included in order to completely understand its advantages over its competitors. At the end of this chapter, Section 1.6 gives an overview of

the fault-tolerance tool introduced in this work.

1.1. Approaches for parallel fault tolerance

This chapter focuses on fault tolerance approaches for MPI applications [31]. The reason is MPI's status as *de-facto* standard. However, there is no real limitation for applying the covered techniques to other frameworks, like PVM (Parallel Virtual Machine) [65].

1.1.1. Using intercommunicators

For applications organized in a master-slave fashion, the failure of a client does not significantly affect the server or the other clients that are able to continue executing properly. This structure is robust because communications are always two-sided, and one of the sides is able to easily detect that the other one has failed. Therefore, it is a good context for introducing fault tolerance.

In MPI, the structure which models this kind of two-sided communications is the *intercommunicator*. Processes in an intercommunicator are organized into two groups, and all communications take place between processes in one group and processes in the other. In master-slave applications, the master manages a set of tasks which are submitted to slaves that carry them out. These slaves return the results to the master when the job is done, and ask for a new task to perform. All communications take place between a slave and the master, and there are no collective communications nor communications between two different slave processes.

In order for the master to detect a slave's failure, it can simply replace the default failure handler `MPI_ERRORS_ARE_FATAL` with a new one, which reassigns the task the faulty slave was performing to a different one. Tasks are usually small, and therefore the slave state at the time of failure can be forgotten, simply starting the task over.

In case the master fails, it is necessary to use any of the other approaches in this section for recovering its state. The use of checkpoint-based rollback recovery (see Section 1.1.4) is usually very efficient, since the master's state consists basically of task specifications, their current statuses, and results of completed ones.

1.1.2. Modifications to MPI semantics

MPI objects and function semantics can be modified so that its behavior is made inherently fault tolerant. For instance, collective communications could behave differently in the presence of failed processes, or certain pieces of data could be redistributed upon failure detection.

One such approach is FT-MPI [24], a fault tolerant MPI implementation that can survive failures, while offering the application developer a range of recovery options. The approach in this case is to extend MPI semantics, substituting the two MPI communicator states (valid or invalid) by an array which can be simplified to {`OK`, `PROBLEM`, `FAILED`}. The introduction of an intermediate state allows FT-MPI to deal with communications in the presence of errors in a fault-tolerant way.

When a non-recoverable error in a process is detected, its communicator needs to be modified, either making it smaller (redistributing ranks), leaving a blank to be filled later, or creating a new process to fill in the blank left by the failed one.

While interesting from a scientific point of view, this approach sacrifices the very reason for MPI existence: the standard. Fault tolerance remains a property of the MPI implementation, and not of the application code, which will behave differently when executed on standard MPI implementations and the semantically modified one.

1.1.3. MPI extensions

Instead of modifying function and object semantics, new ones are introduced to provide fault tolerance. This approach presents the same problem as did semantic modifications, only worse. When extending the MPI interface, the application code will not even compile if using a non-extended MPI implementation. Besides, extending MPI is no better than developing isolated, orthogonal fault tolerance libraries. This is far more interesting, since it may provide fault tolerance capabilities without restricting the communication environment. One example of extensions to MPI is MPI/FT [9].

1.1.4. Rollback-recovery

The most widespread approach to fault tolerance is rollback-recovery. It is based on periodically saving the state of the execution to stable storage, recovering it in case of failure. A system is considered to have been correctly recovered if the externally visible behavior of the distributed system is equivalent to some failure-free execution [62]. Note that it is not required that the recovered state has occurred in a previous execution. Rather, it is sufficient that it might have happened. There are many different approaches to rollback-recovery, each of them very different in the way it implements this basic idea. Although many tools tie themselves to specific systems at the implementation level, a fundamental advantage of rollback recovery is that it is essentially independent of the programming paradigm or the communication system being used. Of course, applying it to message-passing applications implies performing certain tasks to ensure the consistency of the global system, which should not need to be done when dealing with sequential applications. A consistent system state for a message-passing application is described by Chandy and Lamport as one in which every message reflected as received by a process's state is reflected as sent in the corresponding sender process's state [18]. This definition allows for a message to be reflected as sent by a process, but not received by its recipient. This is not considered an inconsistent state, since it might happen in a regular execution. However, the message is considered *in-transit* [22]. If the intended receiver of an in-transit message has failed and is recovered on a system built on reliable communication channels, the rollback-recovery protocol needs to ensure that the message is recreated and delivered to its recipient. If the system is built on non-reliable channels, the failure of the receiver is indistinguishable from the failure of the channel itself, and therefore the delivery of the message will be handled by the communications system.

There are two core approaches to rollback-recovery: checkpoint-based and log-based. The former ensures that the processes' state is recovered correctly and consistently up to the last recovery line, which is defined as the most recent consistent set of checkpoints [52]. However, they do not guarantee that the execution of parts of the code that had already been executed prior to the failure, but after creating the checkpoint used for recovery, will be deterministically re-executed exactly as they originally were. Log-based approaches, on the contrary, create and log to stable storage information about non deterministic events, which allows them to recreate such events after recovery exactly as they played out in the pre-failure execution. These approaches rely on the so called *piecewise deterministic (PWD)* assumption [62]: all

nondeterministic events that a process executes can be identified, and the information necessary to replay each event during recovery can be logged and encoded in tuples called the event's determinant. Although log-based approaches are usually called "message-logging" due to the first systems which proposed and implemented these techniques, the log is not restricted to interprocess communications. More types of events, such as generation of random variables, might be logged depending on the actual range of nondeterministic events covered under the PWD for each different system. This is an implementation issue. Note that sending a message is not a nondeterministic event.

The next section is devoted to checkpoint-based rollback recovery techniques. Section 1.3 covers log-based approaches.

1.2. Checkpoint-based approaches

As explained in the previous section, approaches based on checkpointing are able to correctly and consistently recover an application's state, but they do not rely on the PWD assumption. Therefore, pre-failure execution cannot be deterministically regenerated after a rollback. These kinds of approaches are, therefore, not a good choice when dealing with applications that should preserve their pre-failure behavior unchanged, such as applications with frequent interactions with the outside world.

Checkpoint-based approaches are classified into three categories: uncoordinated, coordinated and communication-induced.

1.2.1. Uncoordinated checkpointing

In uncoordinated checkpointing, each process is allowed to take its local checkpoints independently from others. This has advantages. First, there is no need to comply with any coordination protocol that introduces execution overhead. Second, each process can take its local checkpoints when it better fits its needs. For instance, it can do so when the process predicts it is going to be less expensive. However, finding recovery lines in uncoordinated checkpointing is a difficult task. The approach is susceptible to domino effect [52], and since processes checkpoint independently there is the possibility of creating useless checkpoints, which are those that are never part of a valid recovery line. This approach also needs a fairly complex garbage collection

scheme. These problems are further described in the following subsections.

- Finding valid recovery lines

When a process fails using uncoordinated checkpointing, it must be rolled back to a previous checkpoint. However, doing so might break interprocess dependencies introduced by communications. If no further actions are taken, just simply selecting the most recent checkpoint in the pool will not be enough, since the recovered state may be inconsistent. In order to find out whether a state is inconsistent, the failed process needs information about the global dependencies in the application.

The way uncoordinated approaches collect these dependencies is by piggybacking its status into messages sent to other processes [11]. Processes receiving a message are able to detect that their current state depends on the sender's process state when it sent the message. When a process fails, it asks all other processes to send it their saved dependency information. Using this information, it is able to build a graph for finding a valid recovery line. This involves rolling back other processes as well to recover a consistent state. Two different, equivalent methods for building and analyzing such a graph were proposed [11, 70].

- Garbage collection

Garbage collection is a very important process in uncoordinated checkpointing. Since processes create checkpoints independently, eventually some checkpoint files will be never used again. Precisely, for each process p, all files older than the one in the most recent valid recovery line are obsolete, and can be purged. The same algorithm used for finding valid recovery lines is used for determining which checkpoints can be safely purged. This is an expensive process.

- Domino effect

One serious consequence of uncoordinated checkpointing and the algorithms for finding valid recovery lines is that their very existence is not guaranteed. Since checkpoints are taken in an independent way, situations may occur in which no valid recovery lines can be found, because interprocess dependencies force all processes to roll back to the very beginning of the application execution. One such situation is depicted in Figure 1.1. Each m_i denotes a message being sent, while each $c_{j,k}$

denotes the k-th checkpoint in process j. When p_3 fails, it first considers rolling back to checkpoint $c_{3,2}$. Since message m_8 imposes a dependency on p_2, it would be forced to roll back to $c_{2,2}$. This, due to m_7, would force p_1 to roll back to $c_{1,2}$. The analysis moves forward, and the protocol in charge of finding a valid recovery line discovers that the only valid solution is to roll back all processes to their respective beginnings.

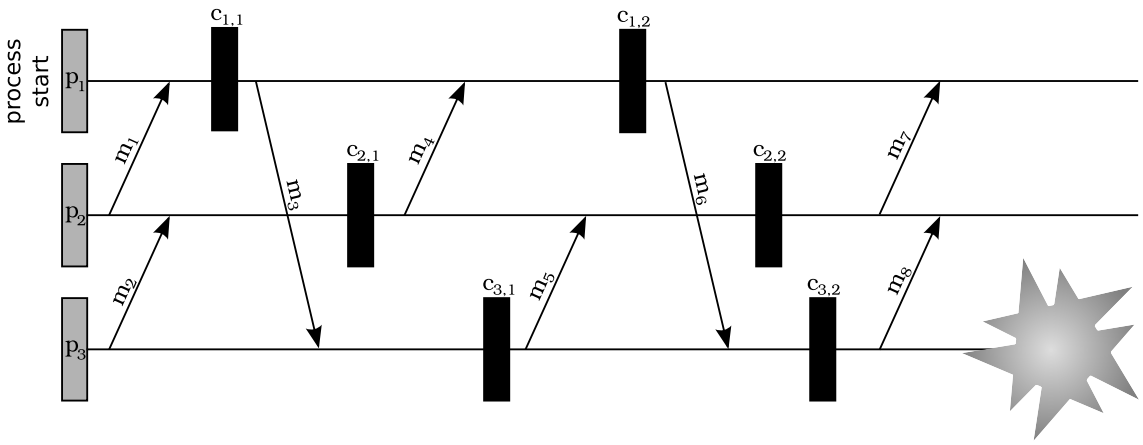


Figure 1.1: Domino effect in uncoordinated checkpointing

1.2.2. Coordinated checkpointing

The idea behind coordinated checkpointing is to ensure that all processes take their checkpoints in a way that ensures that no inconsistent state recovery can take place. There are several ways in which this can be done. The most obvious one is to stop all application progress while the checkpoint is taken [66]. An initiator, which may be an application process or an external entity, sends a broadcast message requesting that processes checkpoint. Upon receiving this message, each process flushes its local communication channels and creates its local checkpoint before resuming execution. This process may lead to significant overheads due to its blocking nature.

In order to reduce the effects of coordination, non-blocking coordinated schemes were proposed. The most studied protocol to date has been the *distributed snapshots* protocol by Chandy and Lamport [18]. The protocol starts with the initiator broadcasting a checkpoint request. Upon receiving the request, each process takes

its checkpoint and rebroadcasts the request before sending any more application messages. The goal is to ensure that processes do not receive any message that could make the global state inconsistent. However, in order for this to be true, the communication channels need to be FIFO and reliable. When working with non-FIFO channels, each process that has just taken its local checkpoint could piggyback the checkpoint request on all further sent application messages, to counter the possibility of the request being received out of order. Another option is to make each process piggyback its current *epoch*[1] in all sent messages. This automatically notifies other processes that a checkpoint has been taken upon receival of a message with an epoch number higher than their own.

If the channel is reliable, coordinated checkpoint techniques ensure that no inconsistencies take place upon recovery. However, in-transit messages still have to be taken into account. If the communications channel is reliable, processes need to log messages received after taking its local checkpoint but sent before the sender took its own. If the communication channel were non-reliable, then in-transit messages are equivalent to messages lost in channel errors, and therefore no further actions need to be taken.

Using coordinated checkpointing, not only the failed processes must roll back, but all processes involved in the computation have to do so as well. This ensures that all events which affected the failed process after the checkpoint was taken are replayed. In this case, finding recovery lines is a very easy operation. Since all processes take their checkpoints in a coordinated way, the last valid recovery line is composed of the latest common checkpoints. Due to this, the garbage collection algorithm is pretty straightforward, and simply consists of removing the oldest checkpoints when newer ones are created.

Another variant involves taking checkpoints based on a coordinated clock [21]. Other authors proposed notifying only those processes which transitively depend on the initiator [35]. The result is that not all processes participate in all checkpoints, but rather, only those that present dependencies among themselves.

[1] In checkpointing terminology, an epoch is the interval between two given checkpoint actions. Therefore, the computation can be divided in a series of numbered epochs.

1.2.3. Communication-induced checkpointing

Communication-induced checkpointing is based on piggybacking dependencies information on application messages so that the domino effect and the creation of useless checkpoints can be avoided. Processes are not coordinated, but rather analyze the state of the global computation they have knowledge of and make decisions about its local checkpoints in two different ways. First, they may take local checkpoints based on their own execution state as in uncoordinated checkpoint approaches. Second, they also take *forced* checkpoints depending on the global dependencies state to avoid the domino effect and the existence of useless checkpoints.

The idea on which communication-induced is based are *Z-paths* and *Z-cycles* [44]. It has been proven that a checkpoint is useless if and only if it is part of a Z-cycle. These approaches piggyback protocol information in application messages so that processes are able to detect when they should take a forced checkpoint to avoid the existence of useless ones. There are two basic approaches: model-based, which detect Z-cycles and force checkpoints to eliminate them; and index-based, which timestamp each local checkpoint on each process and piggyback these timestamps on application messages so that each process decides whether or not to take forced checkpoints, analyzing the timestamps received by other processes. These two approaches have been proven to be fundamentally equivalent [33].

Communication-induced checkpointing is theoretically more efficient and scalable than coordinated checkpointing. However, this technique may indeed create checkpoints that will not be used, and is extremely dependent on the message-passing pattern of the application. This leads to unpredictable checkpoint rates, making the approach difficult to use in practice [23].

1.3. Log-based approaches

Log-based approaches consider the execution of a parallel application as a sequence of deterministic state intervals. Each interval begins with the execution of a nondeterministic event, modeled as the reception of a nondeterministic message. Nondeterministic events include application messages and other forms of nondeterminism such as generation of random numbers or reception of user events. By logging nondeterministic events and relying on the PWD assumption, log-based techniques guarantee that the recovered execution will exactly recreate the nondeterministic

events that had already taken place. Since deterministic events generated by the failed process (e.g. sent messages) will be recreated exactly as they had been in the pre-failed execution, non-failed processes are guaranteed not to become orphaned by the roll back of the failed processes and are never required to roll back themselves, as opposed to checkpoint-based rollback-recovery. This is formalized in the no-orphans consistency condition [7]:

- Let $Depend(e)$ be the set of processes that are affected by a nondeterministic event e.
- Let $Log(e)$ be the set of processes that have logged the determinant of event e in their volatile memory.
- Let $Stable(e)$ be a predicate that is true if e's determinant has been logged to stable storage.

$$\forall e : \neg Stable(e) \rightarrow Depend(e) \subseteq Log(e)$$

This means that, for any event e, if it is not logged in stable storage, then all processes depending on it must have it available in their volatile memory. Indeed, if a process depending on an event e could not access it via stable storage nor volatile log, this process would be an orphan process, since it would depend on a non-recoverable nondeterministic event.

These kind of approaches periodically create uncoordinated checkpoints in order to reduce the amount of re-execution needed upon recovery.

There are basically three types of log-based protocols for guaranteeing that the no-orphans consistency condition is true: pessimistic, optimistic and causal. They are described in the following subsections.

1.3.1. Pessimistic log-based approaches

Pessimistic approaches synchronously log every nondeterministic event as soon as it is generated, not allowing execution to resume until it is available in stable storage. Therefore, they implement a stronger condition than the no-orphans consistency one. It may be logically expressed as:

$$\forall e : \neg Stable(e) \rightarrow |Depend(e)| = 0$$

That is, if a nondeterministic event has not been logged to stable storage, this means no process depends on it.

These approaches have a main disadvantage: the overhead of synchronously logging all received nondeterministic events. In order to reduce this overhead, which makes the approach highly non-scalable, techniques such as using special logging hardware [2] are used. Another option is delaying the logging until the process communicates itself with any other process, an approach known as sender-based pessimistic logging [34]. Depending on the communication patterns of the application, this may log more than one event at the same time, therefore reducing I/O overhead. This may be a problem when using reliable channels. If the process fails before logging the received messages to stable storage, these will not be available to it upon recovery.

In terms of garbage collection, these protocols require only the last stored checkpoint to be kept for each process.

1.3.2. Optimistic log-based approaches

These protocols asynchronously log nondeterministic events [62]. Each process keeps its determinants in a volatile log, which is committed to stable storage periodically. These kinds of protocols have a good failure-free performance, although they do not validate the no-orphans condition. Indeed, if a process were to fail, having determinants in its volatile log not yet committed, all processes depending on events generated by the failed process would become orphans, which requires them to roll back. Thus, optimistic log-based approaches are very similar to uncoordinated checkpointing in case that a process fails without having committed any determinant to stable storage. This results in complex recovery protocols, as well as in a complex garbage collection. Moreover, if eventual stable log of determinants is not guaranteed, the approach is subject to the domino effect.

1.3.3. Causal log-based approaches

These protocols combine the advantages of both pessimistic and optimistic approaches. They are based on piggybacking the contents of each process's volatile log in messages it sends, so the information is available on other volatile logs. Therefore, surviving processes are able to help failed processes replay their pre-failed execution in a deterministic way. By doing so, it is guaranteed that no orphan processes may exist, and therefore no processes are forced to roll back other than failed ones. Their drawback is the communications overhead introduced by the piggybacking of local determinants on sent messages.

1.4. Implementation properties of rollback-recovery protocols

Whether the protocol used is checkpoint-based or log based, all rollback-recovery protocols create checkpoint files as their fundamental recovery unit. It is the only data available to processes in checkpoint-based approaches, and it allows log-based protocols to efficiently recover after a failure.

There are important properties associated with the actual way of creating state files that affect the performance and capabilities of real rollback-recovery tools. We highlight three such properties: granularity, transparency and portability.

1.4.1. Granularity

This property determines the amount of data that the fault tolerance approach stores as part of the checkpoint files. In particular, it determines how much data, compared to the total application data, the technique stores. There are two fundamental approaches to this; full checkpointing, consisting in storing the entire application state, including structures such as the application stack or heap; and variable-level checkpointing, which identifies and selects variables which are needed for restart and stores only those in the checkpoint files. These are sometimes related to the level of abstraction at which the rollback-recovery is implemented. If it is implemented on a system level, that is, on a higher abstraction level than the application itself, it does not have any knowledge of the application internals.

Therefore, a variable-level analysis is not possible, and it must store the application state entirely. Other approaches, implemented at application level, have access to the application code and can modify it in order to introduce fault tolerance. These have access to internal application information and can use it to their advantage. Thus, all system-level rollback-recovery tools must perform full checkpointing, while application-level checkpointers can choose whether to opt for variable-level checkpointing instead.

The fundamental advantage of full checkpointing is that it treats an application as a black box, without knowledge of its internals or previous analysis. It is, therefore, completely transparent for the programmer (see Section 1.4.2), as it can be implemented at the operating system level or even in the hardware. However, it has two important drawbacks. First, the literature has shown that the actual state writing to stable storage is the largest contribution to the overhead of checkpointing [47]. Storing all application data will have a higher associated cost than storing just necessary data. Not only that, but transferring bigger checkpoint files over a network will also have a higher cost. Second, checkpoints will lack portability (see Section 1.4.3). Variable-level checkpointing, on the other hand, is able to obtain better performance and may create portable checkpoint files depending on its implementation. The drawback is the need for complex analyses of the application code in order to identify the state that needs to be stored.

- Incremental checkpointing

An optional improvement in both full and variable-level checkpointing is the use of *incremental* checkpointing [4,25,29,50]. Under this approach, there are two types of checkpoints: full and incremental. Full checkpoints contain all data that is to be stored, while incremental checkpoints store only the data that has changed since the last checkpoint. The recovery process uses the most recent available full checkpoint, and then orderly applies the changes reflected in the incremental ones to completely reconstruct the process state. Creating more incremental checkpoints between each full checkpoint improves the failure-free performance of the approach. Reducing it improves recovery performance. A compromise must be reached, taking into account the failure rate of the system.

When incremental checkpointing is applied to full checkpointing, it is usually done at the page level, using hardware mechanisms to check which pages have changed since the last checkpoint, and must be written in the incremental state

file. When applied at the variable-level, some kind of algorithm to detect changes to individual variables must be applied. Therefore, incremental checkpointing will have a higher impact in terms of runtime overhead when applied to variable-level systems.

1.4.2. Transparency

This property refers to how the user perceives the fault tolerance solution. It is loosely related to granularity, since it might be clear that a full checkpointing scheme will be completely transparent for the users, while a variable-level one will require them to provide application-specific information. However, there are hybrid approaches that do not follow that rule. Some tools perform full checkpointing but still require the user to select where checkpoints must be taken. Similarly, variable-level approaches may employ code analysis and transformations to automatically extract the information, behaving in a transparent way.

1.4.3. Portability

We say a checkpointing technique is portable if it allows the use of state files to recover the state of a failed process on a different machine, potentially binary incompatible or using different operating systems or libraries. The basic condition that has to be fulfilled in order to achieve potential portability is not to store any low-level data along with the process state. Therefore, all full checkpointing approaches are non-portable. Note that not any variable-level approach will be portable, though. Some implementations sacrifice portability for simplicity. For instance, one implementation may store the program counter along with the application data in order to simplify restart.

The second condition for a checkpointer to be portable is that all data is stored in a portable format, so conversions may be made in case they are necessary for recovering the process state on a binary incompatible machine.

1.5. Existing checkpointing tools

A number of tools for checkpointing have been proposed in the literature, each one with its own approach. This section describes the most relevant ones, detailing what they miss, in our opinion, in order to be usable on modern computing frameworks.

1.5.1. CoCheck

CoCheck [61] is a full checkpointing library for parallel applications based in Condor [40]. It uses a coordinated checkpoint-based approach, with messages requesting checkpoints acting as initiators. It works under the assumption of FIFO channels. Upon reception of a checkpoint request, each process stores the state of its communication buffers and sends a request via all its communication channels. Therefore, any in-transit messages will be in the communication buffers which are stored along with the process state, removing the possibility of orphan processes. No inconsistent messages exist since the approach is coordinated, and processes are not allowed to send any new messages until their local checkpoint is completed.

The drawbacks of this approach are its coordinated nature, which hampers its scalability, and the fact that it uses full checkpointing, which makes it unable to work on heterogeneous environments and also results in lower efficiency. Besides, CoCheck depends on *tuMPI*, a specific MPI implementation. Therefore, the code will not be portable to machines without *tuMPI* support.

1.5.2. CLIP

CLIP (Checkpointing Libraries for the Intel Paragon) [19] is focused on checkpointing parallel applications written for Intel Paragon architectures. This is a MIMD multicomputer that uses MPI or the Intel NX libraries for message-passing communications. CLIP implements efficient and simple solutions, but is heavily tied to the Paragon architecture.

CLIP uses full checkpointing. However, it allows for dead memory regions to be excluded from the checkpoint file, which improves efficiency. It requires the user to specify checkpoint locations in the application code. This is required because the

authors did not wish to modify the Paragon kernel, and so they needed to ensure that checkpointing does not take place inside a communication or I/O call.

It implements the same coordination technique seen in CoCheck: all processes are coordinated, FIFO channels are assumed and communication buffers flushed. These are stored at userspace, and communication calls are substituted for CLIP versions which check that the requested message is not on the userspace buffers, and delegate their execution on the original version if it does not find them. This solution, although it is implemented on top of the MPI implementation, does require the library to have knowledge about how the communication buffers are implemented, and is therefore architecture-dependent and not applicable in a heterogeneous environment.

Portability is obviously not a design goal since the architecture is completely Paragon-centric.

1.5.3. Porch

Porch [51] is a source-to-source compiler that translates sequential C programs into semantically equivalent C programs which are capable of saving and recovering from portable checkpoints. The user inserts a call to a checkpoint routine and specifies the desired checkpointing frequency. A compiler then makes source-to-source transformations to instrument the operation. Its data hierarchy consists of a *shadow stack*. This is a copy of the original runtime stack, which is built by visiting each stack frame and saving its local variables, identified at compile time. At the same time, each active function is identified and the call sequence is stored. The computation state is recovered by replaying the original call sequence and restoring each stack frame using the shadow stack's contents. The shadow stack is stored in a portable format, called UCF (Universal Checkpoint Format).

Although this is a sequential approach, it is of fundamental importance in the context of this thesis, since it introduces fundamental ideas and techniques for checkpointing portability.

1.5.4. Egida

Egida [53] is a fault tolerance framework able to synthetize an implementation of the desired rollback-recovery protocol by interpreting a specification language to

express its details. It facilitates rapid implementation of fault tolerance mechanisms for MPI applications. Although it does not depend on any specific MPI implementation, it implements the fault tolerance at the communication level. Therefore, low level changes to the MPI layer are needed before an application can take advantantage of the synthesized modules.

1.5.5. Starfish

In Starfish [5], each MPI node runs a separate Starfish daemon, which communicates with other daemons via the Ensemble group communication system [32]. These daemons are responsible for spawning application processes, keeping track of applications health, managing the configuration and settings of the cluster, communicating with clients, and providing the hooks necessary to provide fault tolerance. Its modular system enables the use of different checkpoint-based protocols, namely uncoordinated and coordinated forms of checkpointing.

Since the checkpointing mechanisms is implemented at the OCaml virtual machine, and given that OCaml is platform-independent, it is able to operate in heterogeneous clusters despite doing full checkpoints. However, the use of OCaml is also the biggest drawback of the approach. Interpreting bytecode does is less efficient than running native code. Doing full checkpoints also hinders performance. Besides, it is limited to host languages interpretable by the OCaml VM.

1.5.6. MPICH-V2

MPICH-V2 [12] is a communications driver for MPICH. It provides transparent fault tolerance by using the Condor checkpoint and restart capabilities. Therefore, it uses full checkpointing with no support for portability and heterogeneous environments.

This is a sender-based pessimistic logging approach. Each process stores a unique identifier and a timestamp for each message it receives. When it needs to rollback, it checks this registry in order to identify the messages that need to be replayed, and asks the original sender to do so. In addition to sending these messages, the sender process knows that the receiver will never ask for any previous message, and performs garbage collection on its sending log.

Although the process is efficient, since it involves no process coordination, the

scalability is hindered by the log-based approach. Performing full checkpointing does not help efficiency nor portability. The fact that it is implemented as an MPICH driver forces all machines to implement it in order to obtain fault tolerance.

1.5.7. MPICH-GF

MPICH-GF [71] is an extension to MPICH-G2 (the Globus communications driver in MPICH) which adds fault tolerance. It is a transparent approach using full checkpointing. It uses coordinated checkpointing by using FIFO channels and flushing communication buffers upon receiving a checkpoint request from the initiator. It implements collective communications through point-to-point actions, which is unefficient. In fact, its overhead figures range between a 10% for the instrumentation and 20% if checkpointing is introduced.

1.5.8. PC^3

The C^3 system (Cornell Checkpoint Compiler) [13–15] is the base for PC^3 (Portable C^3) [26]. The user must insert checkpoint locations and a compiler is in charge of orchestrating fault tolerance through full checkpointing. It is implemented on top of MPI, and therefore does not need any specific implementation to be used in order to obtain fault tolerance.

In order to obtain consistency, it uses a non-blocking coordinated protocol, piggybacking information into sent messages. Since it is on top of MPI, it cannot assume the communications channel to be FIFO, since an application may decide to receive its messages out-of-order. In order to solve this problem it implements a modified version of the Chandy-Lamport algorithm, which is also heavier. Particularly, it converts collective communications into point-to-point ones, further damaging the scalability of the approach.

This approach is currently restricted to C applications.

1.6. Proposal

This work introduces a compiler-assisted checkpoint-based tool for portable checkpointing in parallel environments, called ComPiler for Portable Checkpointing (hence-

forth, CPPC). It is composed of two cooperating systems: a library providing checkpointing routines for parallel applications, described in Chapter 2, and a compiler for automating the code transformations necessary in order to take full advantage of the library, described in Chapter 3. This design combines some of the ideas explored in this chapter with new ones focusing on achieving better qualitative and quantitative results. This section describes its most important properties, techniques and protocols in terms of the properties described in this chapter.

1.6.1. Spatial coordination

As discussed in Section 1.2.1, uncoordinated checkpoint-based approaches present a clear advantage: the lack of coordination overheads. However, they are also subject to the domino effect, and are able to create useless checkpoints, that never become part of valid recovery lines. It would be desirable to retain the efficiency and scalability obtained from not adopting complex runtime protocols while, at the same time, guaranteeing the execution progress in the presence of failures. One way to do so is to use *spatially coordinated checkpointing*. This is a form of coordinated checkpointing that, instead of taking all checkpoints at the same time forming a consistent global state, focuses on SPMD codes and ensures that all checkpoints are created *at the same code locations* in a way that guarantees the consistency of the recovered state.

The basic difference between parallel and sequential applications in terms of consistent recovery is the existence of dependencies imposed by interprocess communications. If a checkpoint is placed in the code between two matching communication statements, an inconsistency would occur upon recovery, since the first one will not be executed. If it is a send statement, the message will not be resent and becomes an in-transit message, also called *in-flight*, *missing*, or *late*. If it is a receive statement, the message will not be received, becoming an inconsistent message, also called *ghost*, *orphan*, or *early*.

The proposal of spatially coordinated checkpointing implies identifying, at compile time, code locations at which the non-existence of in-transit, nor inconsistent messages is guaranteed. An example is shown in Figure 1.2. Let us define a *safe point* as a point in the code were it is guaranteed that no in-transit nor inconsistent messages can exist. Let us assume that:

- All checkpoints are placed at safe points.

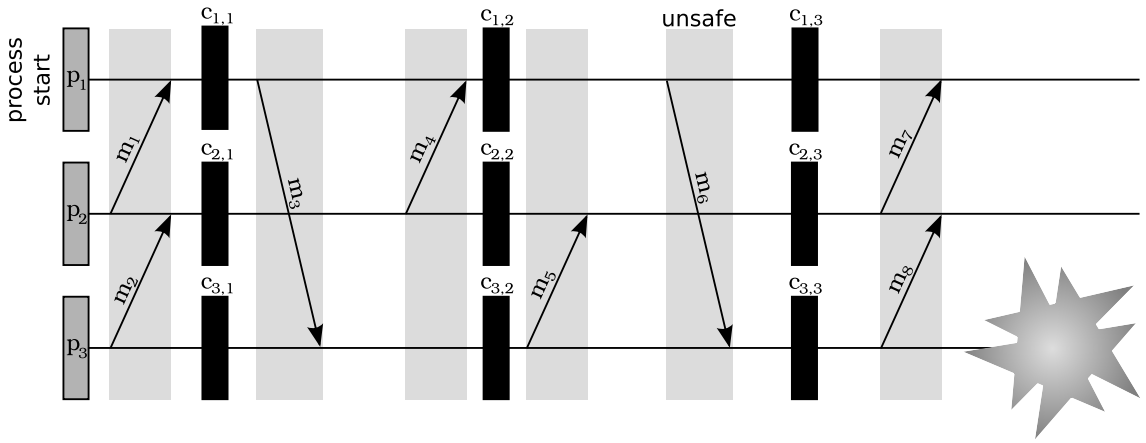


Figure 1.2: Spatial coordination for non-blocking coordinated checkpointing

- All processes take the same number of checkpoints at the same safe points in the code.

It is clear that, under these conditions, all checkpoints will take the same number of checkpoints during their execution. Given two checkpoints $c_{i,x}$, $c_{j,y}$, let us define the *consistency* relation, $c_{i,x} \sim c_{j.y}$, as a binary relation that is true whenever there are no in-transit, nor inconsistent messages between processes i and j between their x-th checkpoint and y-th checkpoint respectively. Then, by the definition of safe point:

$$\forall i, j, x : c_{i,x} \sim c_{j,x} \tag{1.1}$$

Let us define the set of all checkpoints taken by each process i, $\Omega(i)$. Finding a valid recovery line is as simple as finding the checkpoint index x that verifies:

$$\forall i : c_{i,x} \in \Omega(i) \wedge \nexists y / \left(\forall j : c_{j,y} \in \Omega(j) \wedge y > x\right) \tag{1.2}$$

That is, a checkpoint with index x exists in the set of checkpoints available for recovering all processes, and it is the greatest index to fulfill that condition. By Equation 1.1, the set $\{c_{i,x},\ i = 1, \ldots, n\}$ will form a valid recovery line for all processes. According to Equation 1.2, it is also the most recent one. The garbage collection algorithm will be equally simple. Processes can periodically communicate, in an asynchronous way, their most recently created checkpoint. By simply storing

the minimum received value for all processes, a process i may determine that any checkpoint with an index lower than the minimum will not be part of any valid recovery line in the future.

This approach has several advantages: it rules out both the possibility of the domino effect, and the creation of useless checkpoints, without the need for any specific protocol at runtime; checkpoints are taken in a completely scalable way, since the number of processes does not affect in any way how chekpoints are created; no assumptions are made about the properties of the communications channel, which could be unreliable and/or non-FIFO without affecting the protocol.

The places at which checkpoints are to be created are statically determined by the compiler, which needs to perform a static analysis to identify safe points and also an analysis to determine which of the detected safe points are adequate locations for checkpointing. The internals of these analyses are described in Chapter 3.

1.6.2. Granularity and portability

One of the most important design goals for CPPC is portability. With this in mind, the possibility of performing full checkpointing is ruled out, since it is inherently non-portable. CPPC uses a variable level approach, storing only user variables in state files. Furthermore, it only stores live variables, that is, those having values which are necessary for state recovery. By restricting the amount of saved data, checkpoint files are made smaller and so checkpointing overhead decreases. This also improves the performance of network transfers, if necessary.

The use of portable storage formats guarantees a consistent recovery of variable data in binary incompatible architectures. However, upon restart not only user variables need to be recovered, but also non-portable state created in the original execution, such as MPI communicators, virtual topologies or derived data types. This introduces the need for a restart protocol that regenerates the original non-portable, non-stored state. CPPC uses selective code re-execution to achieve complete application state recovery. Therefore, non-portable state is recovered by the same means originally used to create it, making a CPPC application just as portable as the original one: variables are saved using portable formats, while non-portable state is recreated using the original code.

A piece of application code is defined as *Required-Execution Code* (REC) if it must be re-executed at restart time to ensure correct state recreation. The recovery

process consists of the selective and ordered re-execution of such blocks of code. There are six types of RECs, that correspond to:

- Initialization of the CPPC library.
- Portable state recovery: done by the CPPC registration routines.
- Non-portable state recovery: performed through re-execution of procedures with non-portable outcome.
- Restart completion tests: carried out by the checkpointing routine. It ensures that the execution has reached the point where the checkpoint file being used for recovery was originally created, and that the application state has been correctly recovered.
- Conditional sentences: Conditional blocks which contain one or more RECs need to be tested, since not all processes might have executed the enclosed RECs in the original execution. For instance, consider an application that performs file I/O through one, and only one, of the parallel processes. I/O operations will be enclosed in a conditional expression checking the process ID inside the parallel application. At restart time, conditional expressions are evaluated to ensure that each parallel process executes only RECs contained in appropriate conditional branches.
- Stack recovery: Calls to procedures which recursively contain (i.e. through any number of nested procedure calls) one or more RECs. This rebuilds the original application stack.

REC detection is completely automated by the CPPC compiler, which is able to automatically find all described REC types. CPPC controls execution flow when restarting an application, making it jump from the end of an REC to the beginning of the next one, and skipping non-relevant code. The result is an ordered execution of selected state-recovering statements which, eventually, creates a replica of the original application state.

1.7. Summary

This chapter describes the various alternatives for fault tolerance in parallel applications in the literature. It is focused in MPI applications, since it is the

de-facto standard. We identified rollback-recovery as a better approach to fault tolerance than MPI extensions and modification of MPI semantics, for it preserves the MPI interface and adapts better to modern heterogeneous environments. The two core approaches to rollback-recovery, checkpoint-based and log-based, have been described, along with three fundamental properties of checkpoint file creation: granularity, transparency and portability. Then, existing checkpointing tools have been analyzed and categorized, outlining the drawbacks they present in the areas of scalability, efficiency and portability. As long as each of them has its unique advantages, none manages to properly address all three problems at the same time. Finally, CPPC (ComPiler for Portable Checkpointing), the checkpointing approach presented in this work, has been described, outlining its most important characteristics and design decisions. It guarantees restart consistency through the use of compile-time coordination, a manner of implicit process coordination that does not require communications between processes to be performed at runtime. It provides a portable operation by employing a restart protocol that enables portable recovery of non-portable objects through selective code re-execution.

From this point on, we describe the implementation of the two fundamental parts of CPPC in greater depth: the checkpointing library, and the source-to-source compiler.

Chapter 2

CPPC Library

This chapter discusses the design and implementation details for the CPPC checkpointing library, which provides all necessary routines for checkpointing. It is implemented in C++, and its design follows the model-view-controller (MVC) [16] design pattern in a way that allows the same core implementation to be used by different host programming languages. The model is structured in layers [16], decoupling each functional level from the rest. The following sections provide an in-depth description of the library features, following a top-down approach that begins at the application code (the view in the MVC design), describing what services the library provides and how it provides them, while traversing deeper into the framework design, covering the controller and how it adapts external requests to the internal model and, finally, presenting the model and each of its layers.

2.1. View

In our MVC design, the view corresponds to the code of a parallel application that uses CPPC in order to obtain fault tolerance. As in any MVC design, the view is interchangeable so that the framework can service different contexts. For CPPC, this means allowing the same model to be accessed from different programming languages. Specifically, interfaces for C and Fortran 77 have been implemented, but specifying interfaces for any language supporting calculated gotos[1] is straightforward. The need for calculated gotos is justified later on, and exists due to restart

[1] A calculated goto is a goto whose target is an array-indexed address.

protocol implementation details (see Section 2.1.7).

Since the number and semantics of functions exposed to the view is constant, they can be explained without referring to any specific implementation, except for illustrative purposes, in which cases we default to the C version of the interface. The following subsections describe the services CPPC provides to parallel applications through its API, briefly summarized in Table 2.1. CPPC has two operation modes: *checkpoint operation* and *restart operation*. The checkpoint operation mode is used when a normal execution is being run. It consists of marking relevant variables (*variable registration*) and saving them into state files at checkpoints in the code. Restart operation mode emerges when the application must be restarted from a previously saved checkpoint file. Library functions will behave differently depending on the current execution state.

2.1.1. CPPC initialization and shutdown

CPPC needs initialization operations to be performed before servicing most application requests. The initialization operations must be spread amongst two different function calls. The first, `CPPC_Init_configuration()`, reads configuration parameters. Amongst others, it decides whether a restart is to take place. It should be executed as soon as possible, so it can process and remove all CPPC command-line parameters that should not become visible to the application itself, to avoid syntax errors. In its C version, this call receives the command-line parameters and their number, much like `MPI_Init()` does. After its execution, the library knows whether or not to enable the restart protocol. If so, selective code re-execution eventually directs the execution flow towards the `MPI_Init()` call and, after it, to the `CPPC_Init_state()` call. This second initialization function creates necessary memory structures and, if a restart is taking place, calculates the recovery line and makes restart data available to all parallel processes. Interprocess communications are required in order to find the recovery line. Thus, the `CPPC_Init_state()` function must be called after the MPI subsystem has been initialized. This is the reason to spread the initialization operation into two different function calls.

As for the finalization of the library, the `CPPC_Shutdown()` function must be called to ensure a consistent shutdown. Besides removing unnecessary files and freeing memory, this call ensures that any multithreaded checkpoint operation finishes before allowing the program to exit.

Table 2.1: Interface summary of the CPPC library

Function	**Checkpoint operation**	**Restart operation**
`CPPC_Add_loop_index()`	Notifies CPPC that the execution is about to enter an instrumented loop	
`CPPC_Commit_call_image()`	Stores parameter values	Does nothing
`CPPC_Context_pop()`	Notifies CPPC that a function has returned	
`CPPC_Context_push()`	Notifies CPPC that a function is being called	
`CPPC_Create_call_image()`	Notifies CPPC that a non-portable call follows	
`CPPC_Do_checkpoint()`	Dumps state file	Checks restart completion
`CPPC_Init_configuration()`	Initializes configuration	
`CPPC_Init_state()`	Initializes state	Initializes state Finds recovery line Reads checkpoint file
`CPPC_Jump_next()`	Returns 0	Returns 1
`CPPC_Register()`	Creates new register	Recovers data Recreates register
`CPPC_Register_descriptor()`	Creates new file descriptor register	Recovers descriptor state Recreates descriptor register
`CPPC_Register_parameter()`	Stores parameter data	Recovers parameter data
`CPPC_Remove_loop_index()`	Notifies CPPC that the execution has exited an instrumented loop	
`CPPC_Set_loop_index()`	Notifies CPPC that the execution has started a new iteration in an instrumented loop	
`CPPC_Shutdown()`	Waits for checkpoints to complete Frees memory used by CPPC	Not used
`CPPC_Unregister()`	Deletes existing register	Not used
`CPPC_Unregister_descriptor()`	Deletes descriptor register	Not used

2.1.2. Variable registration

CPPC follows a variable level approach to checkpointing. This means that the application code must explicitly mark the variables that are to be stored in state files. We call this process **variable registration**. The function used to register a variable is `CPPC_Register()` which, in its C version, receives the following parameters:

- Base address: The base memory address of the variable to be registered. For C scalar variables, the address-of operator (`&`) should be used.
- Number of elements: How many elements of type specified through the third parameter are being registered.
- Data type: Since portability is one of the top CPPC priorities, stored data needs to be labelled in a way that allows for conversions to be performed if necessary. Besides, this labelling should be independent of the host programming language or the communications system used (MPI provides its own data type codes). Thus, CPPC defines constant values associated to the basic data types to be registered. For instance, registration of character types is done using `CPPC_CHAR`, `CPPC_INT` is used for integers, etc. When implementing views for languages supporting overloaded methods the use of labels would not be necessary. Instead, a different function could be used for each data type.
- Register ID: The base address is generally not unique for different variables (e.g. memory aliasing in C). Thus, each register is uniquely identified by a string. Any string that is unique for each function scope is valid. Therefore, the use of the variable name is recommended.
- Memory type: A boolean value, indicating whether the memory to be registered is static (scalar variables and arrays) or dynamic (pointers). This is relevant when recovering the variable value, as will be further explained. This parameter is not present in languages such as Fortran 77 (which uses static memory only) or C++ (where partially instantiated [6] template functions [63] can be used to detect the difference).

The register function behaves in a different way when the execution is being recovered from a state file. If called in this situation, it will not only perform the registration, but also recover the original data found in the checkpoint file. When processing static variables, these data are copied to the memory allocated for them

by the compiler. For dynamic variables, the function returns a pointer to a memory address containing the data. For instance, assume that an array containing 100 integers named "data" is to be registered in a C application. The registration call syntax would be:

```
CPPC_Register( data, 100, CPPC_INT, "data", CPPC_STATIC );
```

In a regular execution, the address of the `data` array is registered for future storage, containing 100 integers (the actual size in bytes depends on the computing platform and is calculated internally). When the application is restarted, calling this function recovers the original array values by copying them into the compiler-allocated memory. Also, as in a normal execution, a register for the variable is created. If `data` were not an array, but a pointer containing 100 integers, the registration call would be:

```
data = CPPC_Register( data, 100, CPPC_INT, "data", CPPC_DYNAMIC );
```

Where the return value to the variable, and the memory type parameter indicates that this register involves dynamic memory. Upon application restart, the call returns a memory address containing the original data, instead of replicating them into the allocated memory.

A `CPPC_Unregister()` function is also provided for the removal of obsolete variable registers. Variables that have lost relevance are excluded from future checkpoints, thus optimizing state files sizes.

2.1.3. Non-portable calls

As seen in Section 1.6.2, CPPC uses selective code re-execution in order to recover non-portable state. Function calls having a non-portable outcome are re-executed when recovering an application using the same parameter values as in the original execution. The CPPC library provides functions that store parameter values in order to allow for proper re-execution. Before performing a non-portable invocation, a call to the `CPPC_Create_call_image()` function is issued. The parameters for this function are its line number in the code and the name of the function itself. This pair uniquely identifies a specific non-portable call. Note that simply using the line

number as identifier would not suffice, since a line in the code may contain more than one non-portable call (e.g. by using the return value of a non-portable call as a parameter for another non-portable call). Internally, CPPC creates a frame for the parameters to be stored, named *call image*.

After creating the call image frame, parameters that are required to preserve their values upon restart are passed to a special version of the registration function, called `CPPC_Register_parameter()`, which receives the same parameters as `CPPC_Register()`. The difference between both is that a registered parameter is associated to a call image, rather than to a procedure scope. After registering all required parameters, the `CPPC_Commit_call_image()` function stores the call image for inclusion in subsequent checkpoint files. A typical example of use for instrumenting an `MPI_Comm_split()` call would be:

```
CPPC_Create_call_image( "MPI_Comm_split", line_number );
CPPC_Register_parameter( &color, 1, CPPC_INT, "color",
  CPPC_STATIC );
CPPC_Register_parameter( &key, 1, CPPC_INT, "key",
  CPPC_STATIC );
CPPC_Commit_call_image();
MPI_Comm_split( comm, color, key, comm_out );
```

When restarting the application, the execution flow is directed towards this REC, and the values for `color` and `key` are recovered. Note that the original communicator `comm` is not registered, on account of it being a non-portable object. Either its value is recovered by a previous execution of a non-portable call, or it is a MPI-defined constant, such as `MPI_COMM_WORLD`. Eventually, the non-portable call is executed in the same conditions as in the original run, thus having the same semantic outcome: a new communicator `comm_out`, containing the processes in `comm` with the same value of `color`, ranked attending to the values of `key`. Note that the specific MPI implementation used for restart could be different from the one used in the original run, and the outcome would be semantically correct in the new execution environment.

The basic functional difference between a regular variable register and a call image parameter is that in the former the variable address is saved and its contents stored when the checkpoint function is called. The value of call image parameters, however, is stored in volatile memory when `CPPC_Commit_call_image()` is invoked,

and included in all subsequent checkpoint files. There is no way to remove a parameter registration, since a non-portable call never stops being relevant to the execution in CPPC's scheme.

2.1.4. Context management

The previous sections showed some of the data that CPPC includes in checkpoint files, namely variable registrations and call images. However, in order to correctly categorize each recorded piece of data into its correct scope, the CPPC library must be able to keep track of execution context changes. In order to do so, it internally manages a context hierarchy. Each context object represents a call to a procedure, and contains the information required for recovering data in that procedure scope. Contexts contain variable registers, call images, and also other subcontexts, created by nested calls to the same or other procedures. This hierarchical representation allows for the sequence of procedure calls made by the original execution to be recreated upon restart. In this way, the application stack is rebuilt, and the relevant state is recovered inside its appropriate scope. This hierarchical representation allows for the application stack to be rebuilt upon restart, by recreating the sequence of procedure calls made by the original execution, while recovering the relevant state inside each procedure scope. This heap-shaped hierarchy generalizes any pattern in procedure calling, including recursivity. Two CPPC library routines, `CPPC_Context_push()` and `CPPC_Context_pop()`, are used to notify context changes. These calls are only inserted before and after a call to a CPPC-instrumented procedure, respectively. A CPPC-instrumented procedure is any procedure containing CPPC code, be it registers, call images, checkpoints, or simply calls to procedures containing any of these.

There is another situation that requires tracking by the library, and that is the insertion of non-portable calls inside loops. If this happens, the non-portable call will be repeated a certain number of times, and a different call image must be created for each invocation. Moreover, CPPC must provide means to re-execute the non-portable section of the loop when recovery takes place. For this purpose, the `CPPC_Add_loop_index()`, `CPPC_Set_loop_index()` and `CPPC_Remove_loop_index()` functions are provided. Their purpose is to create a special type of context, called a loop context, which may only contain call images. Other types of registration performed while inside the loop will be automatically inserted into the first predecessor node that represents a regular execution context. The following code is

a simple example of use of this type of context:

```
CPPC_Add_loop_index( "i", CPPC_INT );
for( i = 0; i < n; i++ ) {
  CPPC_Set_loop_index( &i );
  /* non-portable calls go here */
}
CPPC_Remove_loop_index();
```

Loop contexts need only be inserted on loops containing non-portable calls. Regular variable registrations, checkpoints, or other CPPC operations potentially placed inside a loop do not need any special modifications to their environment.

2.1.5. Open files

Files are characterized as being random access dataflows, with a state defined not only by their data or path, but also by a pointer to the current position inside the data stream. This position must be recovered when reopening takes place. The `CPPC_Register_descriptor()` function is used to track open files. It receives a unique identifier for the descriptor (an integer), the base address for the descriptor itself, the type of the descriptor being passed, and the path of the file being opened. An example of use:

```
fd = open( path, mode );
CPPC_Register_descriptor( id_number, &fd, CPPC_UNIX_FD, path );
```

During a regular execution, the register descriptor function marks the file as open, so that relevant information about the descriptor, including the position of the stream pointer, will be stored when a checkpoint is reached. When recovering the application, the library re-executes the `open` function and then the register function, which in this context ensures that the stream pointer is correctly repositioned. The `CPPC_UNIX_FD` parameter tells the library that a regular UNIX file descriptor, represented as an integer, is being registered. Another valid value for the descriptor type is `CPPC_UNIX_FILE`, used for `FILE *` descriptors. More register types can be added to the library by implementing the appropriate handlers and functionalities inside the filesystem abstraction module (see Section 2.3.4).

A `CPPC_Unregister_descriptor()` function is also provided to remove descriptors after their associated file has been closed, effectively removing any further reference to it on future checkpoint files.

2.1.6. Checkpoint file dumping

As mentioned in the previous chapter, calls to the checkpoint function, `CPPC_Do_checkpoint()`, must be introduced at safe points, where no in-transit nor inconsistent messages between processes exist. From the application's perspective, this call simply receives a unique integer parameter which is used to identify the point in the code where the state dumping is performed, so that it is distinguishable from other checkpoints performed in the same execution context, allowing the correct identification of the restart point during the recovery process.

2.1.7. Application restart

The restart phase has three fundamental parts: finding the recovery line, reading the checkpoint data into memory, and recovering the application state. The first two steps are encapsulated inside the `CPPC_Init_state()` call. However, the actual reconstruction of the application state and repositioning of control flow must be reached through the ordered execution of the RECs in the application (see Section 1.6.2). For this purpose, jump labels are inserted into the appropriate positions in the code, and a structure indexing these labels is defined. In the C interface, this structure is an array of ordered jump addresses (`void *` values) for the different labels. The restart process consists of jumping from the end of an REC block to the beginning of the next one, executing its code and jumping again. This process is repeated until the library determines that the restart has finished and that a regular sequential execution of the code can be enabled. The C function the application uses to query for the restart state is `CPPC_Jump_next()`. It returns 0 if no restart is being performed, and a different value otherwise.

Note that jumps must be performed inside the application code without leaving the current procedure scope. Otherwise, the call stack would be inconsistent for the target scope. Also note that there is no more elegant solution than the use of gotos, since RECs can be found at any point or nesting level inside the code, discarding the use of `switch` constructs. In this situation, the use of arrays indexing local jump

addresses is a safe and coherent solution, as long as it is guaranteed that said arrays are not passed as parameters between functions, which CPPC does not do.

Jumps used to reach a REC from the end of another one are called *conditional jumps*, since they are taken inside an `if` structure controlled by `CPPC_Jump_next()`. Their code is as follows:

```
if( CPPC_Jump_next() ) {
   int jump_counter = cppc_next_label++;
   goto *cppc_labels[ jump_counter ];
}
```

Jump labels and conditional jumps are inserted enclosing all the six types of RECs: library initialization, portable state recovery (variable registration), non-portable state recovery (execution of calls with non-portable outcome), restart completion tests (done by the checkpoint function), conditional sentences and stack recovery (execution of calls to CPPC-instrumented functions). For some of these REC types, however, the control flow instrumentation varies slightly. The `CPPC_Init_configuration()` call starts the jump sequence and is usually the first statement in the code, and therefore does not include a jump label before it. Also, conditional sentences and stack recovery require a more complex version of the control flow instrumentation, as detailed in the following paragraphs.

Calls to CPPC-instrumented procedures must be re-executed during restart. For this purpose, `CPPC_Context_push()` and `CPPC_Context_pop()` are placed enclosing the call, which is itself executed as well. Internally, control flow code is added as previously explained, but with the first statement in the procedure being a conditional jump towards the first REC inside its code, and with the last conditional jump reaching a generic return instruction. Note that the generic return does not affect the state recovery process. If the returned value is stored into a relevant variable, then this variable will be registered after the call, and its correct value recovered. Otherwise, the returned value is discarded or stored into a non-relevant variable. In both cases, the generic return cannot affect the execution outcome.

Adding control flow code to an instrumented conditional sentence is similar to doing so to instrumented procedures, just more complex. The conditional expression must be re-evaluated to ensure that each parallel process executes the proper branch. Variables involved in the computation of the control expression are registered as if

they were used in a non-portable call, using the call image-related functions previously detailed. Thus, it is guaranteed that the evaluation of the control expression is consistent with the original one. Each conditional branch is then modified so that its first statement is a conditional jump towards the first REC in that branch. At the end of each branch, a conditional jump is placed that directs the process to the next REC outside the conditional construct. These particular conditional jumps are different from the regular ones in that they do not increase the jump counter by one, but by an amount dependent on the number of RECs between the current position and the desired target. For instance, the first conditional jump in the `then` branch of an `if` statement will increase the jump counter by one. However, the last conditional jump in that branch will increase the counter by the number of RECs contained in the `else` branch plus one. Similarly, the first jump in the `else` branch increases the counter by the number of RECs contained in the `then` branch plus one, while the last one increases the counter by one. This example can be easily generalized for other conditional constructs, such as `switch`.

The presented restart protocol fulfills the portability objective in CPPC. It does not directly modify non-portable state such as the program counter or the application stack, but instead uses the application code itself to perform required modifications using workarounds. It does not impose non-portable constraints such as recovering the variables in the same memory addresses as in the original run. It also is implementable on a wide range of programming languages. Although required code modifications may seem cumbersome and nontrivial, they are all automatically performed by the CPPC compiler described in the next chapter.

2.2. Controllers

To connect the different views with the library model, CPPC's API is implemented into controllers that abstract the particularities of the chosen programming language and translate the requests to the set of C++ functions implemented by the model. Each controller performs different actions depending on its target language. This section covers the controller connecting the model with C views. Given the similarities between this language and C++, the controller is extremely simple. The three fundamental remarks about its implementation are:

- For connecting C and C++, the taken approach is to implement the controller in C++ declaring their functions as `extern "C"`. Thus, the compiler generates

C function signatures, which may be linked with the parallel application.

- Inside the functions declared as `extern "C"` the controller interacts with the model façade. Said façade is a *singleton*[2] that exposes a series of operations almost identical to those seen in Table 2.1. The controller perfoms type conversions, such as converting C-style strings (`char *`) to the C++-style `std::string`.

- Since repositioning a file stream pointer is an operation which depends on the host programming language, it cannot be implemented inside the model. Instead, controllers are responsible for this task. The C version of the controller keeps track of the set of registered file descriptors and, upon restart, gets the appropriate file offset from the model and repositions the pointer stream using the filesystem abstraction interface found in the portability layer (see Section 2.3.4).

2.3. Model

The core implementation of the CPPC library functionality is found inside the model. Figure 2.1 offers an overview of the framework design. The model is structured in layers which decouple each functional level from the rest. It offers a façade which serves as centralized access point, implementing one function for each of the ones exposed in the interface. Under it there is a checkpointing layer, which receives a checkpoint file in a specific format and implements protocol-dependent operations. Then it passes the data to the writing layer, which is an interchangeable piece of software in charge of writing and reading checkpoint files to and from stable storage. There are two additional layers: a portability layer, which abstracts operations which depend on operating system services; and a utility layer, which provides implementations of useful operations, such as data compression or integrity checks, which are commonly used by the other layers. All these are described in subsequent subsections. Note that, at this level, the library code is completely independent of the controller (C, Fortran 77, etc.).

[2] In object-oriented programming, a *singleton* is an object programmed using a design pattern that characterizes a class by exposing a single access point to its instances. This access point must guarantee global accessibility and the existence, at any given time, of a single instance of the class, which is returned in each invocation. There are many variations of this pattern depending on the goals to be achieved. For more information see [6, 28].

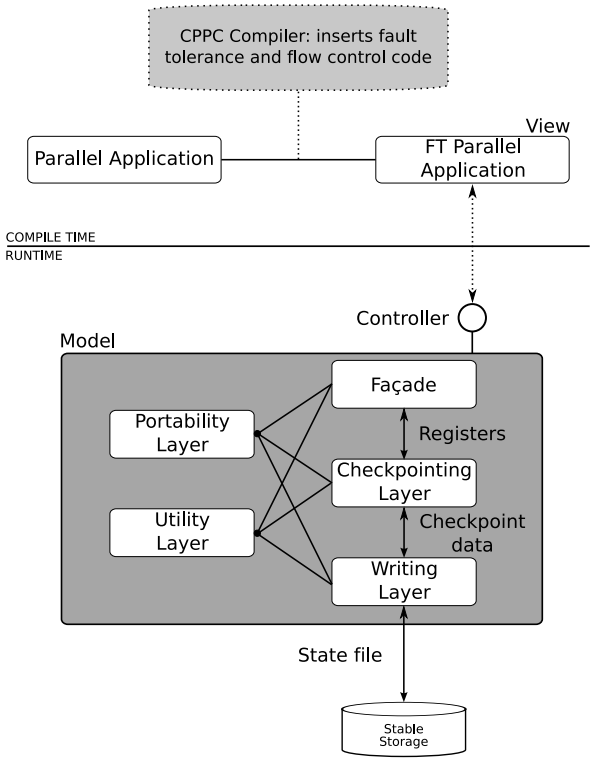


Figure 2.1: Global design of the CPPC framework

2.3.1. Façade

This layer orchestrates all the model behavior, while offering an external interface to the controller. It implements the following functions:

- `CPPC_Init_configuration()`

This function initializes the library configuration. CPPC implements a configuration manager that centralizes the access to different configuration sources into a single interface that abstracts this functionality. Due to the modular design of CPPC, the naming convention for configuration parameters is hierarchical, each level referring to a specific module or layer. How parameter names are constructed depends on the source of each of them, but the first level is always `CPPC`. The configuration manager adopts the following rules for parameter naming:

1. Command-line parameters: hierarchical levels are separated using the "/" char-

acter. For instance, the parameter controlling whether a restart is to take place is named `CPPC/Facade/Restart`.

2. Environment parameters: Environment variables cannot contain the "/" character. Thus, "_" is used for separating hierarchy levels. The restart parameter as defined by an environment variable is called `CPPC_Facade_Restart`.

3. Configuration files: A convenient way to specify parameters which remain constant through many executions. A simple DTD[3] has been created to describe an XML scheme for configuration files. There are basically two types of tags: variables, which have a name and a value; and modules, which may contain variables and other modules. A simple example of configuration file defining the restart parameter would be:

```
<cppc>
  <module name="Facade">
    <variable name="Restart" value="false"/>
  </module
</cppc>
```

The library is also able to accept configuration files in plain format in case no XML parser is available in the computing platform. In this case, the naming convention is the same as that of command-line parameters, and a simple file containing the restart parameter would look like:

```
CPPC/Facade/Restart=false
```

Some common configuration parameters are described in Table 2.2. However, since configurable plugins having their own configuration options may be dynamically added to the CPPC library, this is not a comprehensive list.

- `CPPC_Init_state()`

Once CPPC has been configured, it can initialize its working environment. First, it creates a directory for state file storage. To support shared file systems (commonplace in clusters), a directory tree is created:

[3] A DTD, or Document Type Definition, is an XML schema language primarily used for the expression of a schema via a set of declarations that describe a class, or type of document, in terms of constraints on the structure of that document.

Table 2.2: CPPC configuration parameters

Module	Parameter	Description
Facade	RootDir	Directory where CPPC will store and fetch checkpoint files.
	ApplicationName	Subdirectory of "RootDir" where checkpoint files will be effectively stored.
	Restart	Controls whether the application must be restarted. It is recommended to set this to "false" and enforce restart using command-line parameters.
	Frequency	Number of calls to `CPPC_Do_checkpoint()` between effective state dumping.
	CheckpointOnFirstTouch	If set to "true", the state dumping will be performed every first time that a particular call to `CPPC_Do_checkpoint()` is reached.
	Suffix	Suffix for the generated files.
	StoredCheckpoints	Maximum number of checkpoint files stored per node before beginning deletion of older ones.
	DeleteCheckpoints	If set to "true", stored checkpoints will be removed upon successful execution.
Compressor	Type	Type of compressor used. See Section 2.3.5.
Writer	Type	Type of writer used. See Section 2.3.3.
Checkpointer	Multithreaded	Use multithreaded dumping. See Section 2.3.2.

- Level 1. One folder to store all CPPC applications data. E.g. `/tmp/CPPC_Files`.
- Level 2. Inside the first level, a folder for each application. The name of this folder is determined by the application identifier defined through the `CPPC/Facade/ApplicationName` parameter seen in Table 2.2. For instance, an application named `DBEM` would find its files under `/tmp/CPPC_Files/DBEM`. Automatically determining the application name using the first command-line parameter was discarded, since:
  - Executing an application on queues makes the application name variable depending on its PID[4]. Thus, an application being re-executed with a different PID will not find old checkpoint files created during a regular execution. Besides, if the aforementioned problem did not exist, two different executions of the same application would try to share the same folder name, leading to consistency problems.
  - Not all programming languages accept comand-line parameters. Fortran 77, for instance, does not define a standard for accessing them (although most compilers provide their particular way to do so).
- Level 3. Finally, a folder per process. Each one of them may obtain a unique identifier in the context of the parallel application execution as their rank[5] in the `MPI_COMM_WORLD` communicator. Thus, a process with rank 4 executing a CPPC application would store its files under `/tmp/CPPC_Files/DBEM/4/`.

If it is not necessary to restart the application from a previously created checkpoint (depending on the `CPPC/Facade/Restart` parameter), the initialization process has finished, and the control flow returns to the parallel application. Otherwise, a valid recovery line has to be found.

To determine which files should be selected for restarting, each process must check which checkpoint files are available to it, and communicate with the rest to find the most recent file common to all processes. Since it is guaranteed that all processes take the same number of checkpoints, files with the same name[6] in different processes correspond to the same point in the execution of the application. Since no in-transit nor inconsistent messages existed upon creation, the set of homonym checkpoint files

[4]PID, or process identifier, is a number used by some operating system kernels (such as that of UNIX, Mac OS X and Windows NT) to uniquely identify a process.

[5]MPI assigns each process an integer which uniquely identifes it in a communicator.

```
A = Available state files, ordered by
   unique file code

agreement ← false
While not agreement
   N ← Newest correct state file in A
   If all processes propose N then
      agreement ← true
   Else
      O ← Older restart point proposed
      NO ← {x ∈ A / x is newer than O}
      A ← A-NO
   End If
End While

Delete files older than N
```

Figure 2.2: Pseudocode of the algorithm for finding the recovery line

forms a strongly consistent global state. A pseudocode of the algorithm for finding a valid recovery line is shown in Figure 2.2.

Once the recovery line has been found and each process knows which file it must use for restart, it reads its contents into memory. These data are ready to be recovered as the application progresses through its RECs.

- `CPPC_Shutdown()`

This function finalizes the library, leaving it in a consistent state. First, it ensures that the model being shut down was previously initialized through calls to the initialization functions. Then, if that first check succeeds, it notifies the inferior checkpointing layer that a shutdown is taking place, so that it may take appropriate action. This may involve waiting for any multithreaded checkpoint operations to finish before proceeding. After that, it evaluates the `CPPC/Facade/DeleteCheckpoints` configuration parameter. If set to `true`, it removes all generated checkpoint files, and begins traversing the directory hierarchy built by `CPPC_Init_configuration()`, removing all empty directories, until it finds a non-empty one or removes the hierarchy

[6] CPPC assigns file names incrementally, beginning at 0, and increasing for each newly created checkpoint file.

root. Since the filesystem interface is not standard in C++ (classes for file access are provided, but not for folder manipulation), CPPC includes an abstraction layer for filesystem access to facilitate the portability of the library. The implementation of this layer is operating system-dependent. For more details, see Section 2.3.4. After these steps have been taken, the façade instance is destroyed, library memory freed, and the application is ready to exit.

- `CPPC_Register()`

This function is in charge of variable registration: scheduling memory regions to be stored in subsequent checkpoints. This function behaves in a different way if called during a regular execution or during recovery.

During a regular execution, the façade is responsible for managing the set of registers performed by the application. Thus, when this function is called, a new register is added to this set as a structure containing its base memory address; the end memory address; the data type, which may be necessary in order for inferior layers to correctly store it; and the register name. If the name conflicts with another one in this execution context, the register will not be created. For this reason, as previously stated, the use of the variable name is recommended (no two variables with the same name may exist in the same procedure scope).

In order to calculate the end memory address from the base address, the data type and the number of elements of the register (which are the parameters provided to the function), it uses another of the services provided by the portability layer, in this case a data type manager, that through partial template instantiation is able to provide a single C++ implementation to functionalities such as obtaining the size of a data type, independently of the architecture on which the library has been compiled.

If the application is being restarted, however, it is necessary to recover the data originally stored in the registered memory and restore them to the place where the application expects to find them. Since the checkpoint file contents were recovered when `CPPC_Init_state()` was called, the only thing to be done at this point is to make said contents reachable for the application. If the variable being registered is static, the library copies the stored data to the base address provided. Otherwise, it returns a pointer to the appropriate memory address, that the application may assign to the registered pointer variable. After data recovery is finished, the library

recreates the old register as a regular execution of this call would.

- `CPPC_Unregister()`

This is one of the simplest functions in the façade. It receives a register name and removes the corresponding match from the set of existing registers. Note that the memory address does not uniquely identify a register, since aliasing between variables may cause different registers to start at the same memory address.

- `CPPC_Create_call_image()`

This function call creates a new call image object for inserting required parameter registrations. During a recovery, the function also fetches the old call image from the state file in order to subsequently recover the aforementioned parameters' values.

- `CPPC_Register_parameter()`

At the façade level, this function does not differ much from the regular variable registration. The only difference is that the register is inserted inside the object created by the last `CPPC_Create_call_image()` function, instead of being associated to the current execution context. During restart, the function reads the register from the recovered call image and recovers the parameter value.

- `CPPC_Commit_call_image()`

When a normal execution is run, the commit function performs aliasing checks (see the `CPPC_Do_checkpoint()` section) on the registered parameters inside the associated call image and stores their values in volatile memory to be stored at subsequent checkpoints. During restart, it simply removes the old call image object, which should be now empty after all contained parameter registrations have been recovered.

- `CPPC_Context_push()`

Called when the execution is going to perform a call to a function containing CPPC instrumentation. During a normal execution, it creates a new execution context object and adds it to the context hierarchy in the façade under the current one. After that, it assigns the current execution context pointer to reference the newly created one. During restart, the function handles two different homomorphic hierarchies: besides the regular one, the saved context hierarchy is recovered from the checkpoint file and traversed as the execution progresses. The façade maintains two different pointers to current execution contexts: one for the current execution hierarchy, and another one for the one recovered from the state file, which contains all registers, call images, and checkpoints taken during the original execution.

- `CPPC_Context_pop()`

Called when the execution has returned from a call to a CPPC-instrumented function, it updates the execution context pointer to reference the parent of the existing one. Additionally, if the context the execution is leaving is empty (it contains no registers, checkpoints, call images, or subcontexts), it destroys the object, so that it is not saved to subsequent checkpoints. During a restart, this call also updates the pointer to the current context in the hierarchy recovered from the state file.

- `CPPC_Register_descriptor()`

This function adds an entry to the set of open files to be recorded at subsequent checkpoint files. It does not perform any kind of stream pointer positioning at restart, which is the responsibility of the view as previously explained.

- `CPPC_Unregister_descriptor()`

Removes an entry from the set of current open files.

- `CPPC_Do_checkpoint()`

This is another function with a behavior that depends on whether a normal execution or a restart is being done. In the first case, its purpose is to initiate the

state file creation. There are some configuration parameters that affect the outcome of this operation. First, it is possible to define a checkpointing frequency. It is specified as the number of calls to the `CPPC_Do_checkpoint()` function before an actual checkpoint takes place. If `CPPC/Facade/Frequency` equals `N`, then the state file dumping will only be performed each `N` calls to this function. This parameter is intended to be used in computational loops containing checkpoint calls.

Alternatively, it is possible to define another parameter that forces checkpoints to be taken the first time the execution reaches them. This option is included to avoid skipping checkpoints outside computational loops. These are usually placed at inflection points in parallel programs, where a task has been finished and a new one is going to begin. Usually, after a computation phase is finished its partial results are available to be stored while auxiliar variables used for intermediate results are not relevant anymore, thus reducing state file sizes.

Summarizing, a checkpoint will be taken if: (a) the number of calls is a multiple of the frequency parameter; or (b) it is the first time that a particular checkpoint is reached in the code.

After deciding that a checkpoint file has to be created, the first thing the façade does is to perform aliasing checks on all existing registers. It does so by traversing the context hierarchy structure, starting at its root (the context associated to the main procedure), and analyzing all registers in each context. During this operation, the function creates a list of memory blocks to be stored in the checkpoint file. For each register, it analyzes its beginning and end memory addresses and compares them to the addresses of other registered variables. If no overlap exists, it schedules a new memory block for inclusion in the checkpoint file, corresponding to the memory assigned to the analyzed register. Otherwise the smallest memory block containing all overlapping registers' memory is calculated. Thus, a single memory block in the checkpoint file may be assigned to multiple registered variables. The result of this calculation is the basic checkpoint file format depicted in Figure 2.3. As can be seen, each state file is divided in two different parts: a metadata section and an application data section. The application data section contains the memory blocks copied from the application memory, which hold the relevant data necessary for restart. The metadata section contains information for the correct recovery of the application memory: the execution context hierarchy, containing variable and parameter registrations. The figure omits call images for simplicity, since they are similar to execution contexts, but containing parameter registrations only. For each variable to be recovered, execution contexts store information about its type,

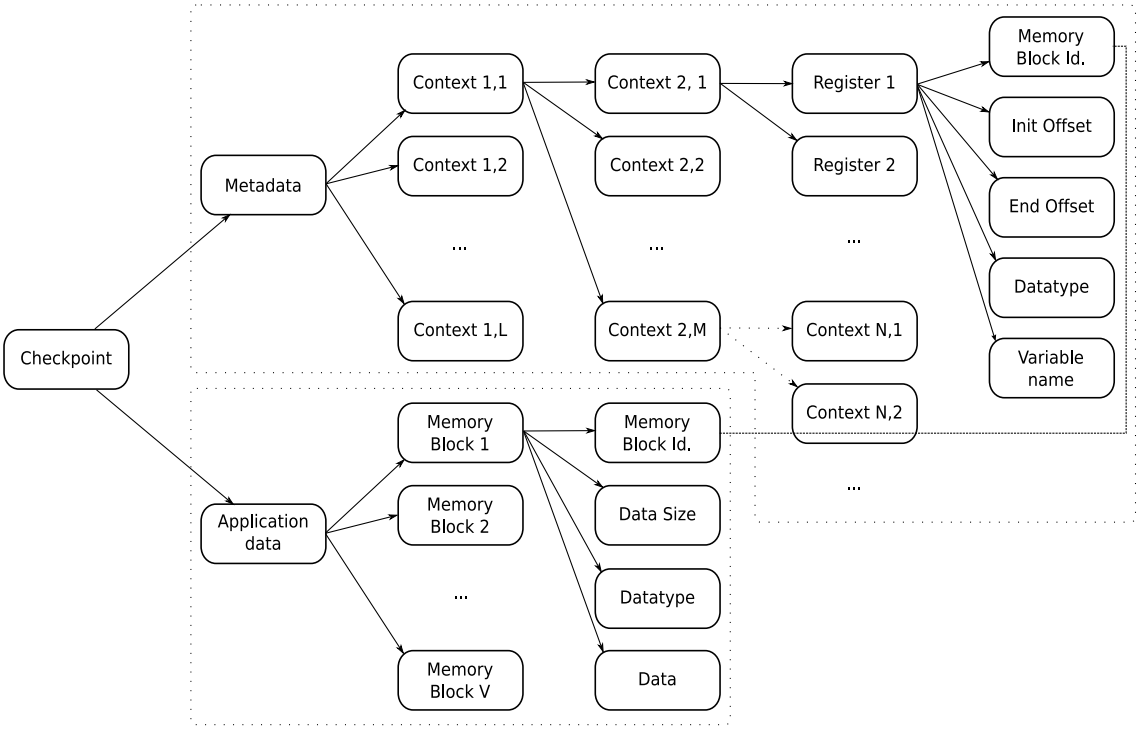


Figure 2.3: Data hierarchy format used for writing plugins

associated memory block in the application data section, and offsets to calculate the beginning and end of its data inside that memory block. The use of portable offsets instead of memory addresses [64] enables pointer portability. The offsets are calculated taking into account the original address for the registered variable and the address for the corresponding block in the application data section. Upon recovery, the memory block will be loaded into memory, and memory addresses for variables to be recovered will be back-calculated using the stored offsets. This preserves aliasing relationships through application restarts.

Note that, while CPPC provides a mechanism for dynamically plugging new writing modules, which perform the data writing to stable storage (see Section 2.3.3), all of them must accept this checkpoint file format as input.

If an execution is being restarted, this function does not perform checkpointing. Rather, the library tests two conditions: (a) that this is the execution context where the checkpoint file used for restart was created; and (b) that this is the exact point inside the procedure code where it happened. If both are fulfilled, the restart is over, conditional jumps are deactivated, and the parallel application will be allowed

to normally resume its execution.

- `CPPC_Jump_next()`

As mentioned when discussing the library views, the purpose of this function is to notify the application as to whether a restart or a normal execution is taking place. It returns a boolean value indicating the current state. The application uses this return value to control conditional jumps.

2.3.2. Checkpointing layer

The façade implements most of its functionality through different sublayers. Precisely, it uses the checkpointing layer and some of the functionalities offered by the portability layer.

The checkpointing layer provides access points for managing the creation of the checkpoint itself. The façade handles, basically, execution contexts, call images, and variable and parameter registrations. When the time comes to include those in a checkpoint file, it uses the services provided by the checkpointing layer, which encapsulates the particularities of the checkpointing protocol itself. This allows for protocol changes without affecting the façade, nor the layers below, either. The services provided by the checkpointing layer are described in the following subsections.

- State file writing

The fundamental goal of this service is to implement the specifics of the checkpoint protocol before delegating the file creation into the layer below. If a coordinated checkpointing scheme were to be introduced, this would be the layer to do so. Since CPPC's compile-time coordination checkpointing scheme does not use runtime coordination or message-logging, there are no relevant actions to be taken at this point.

This service checks that the data passed by the façade are valid, creating a new thread to handle state dumping (multithreaded checkpointing) and return. However, it also needs to ensure that the two threads do not incur inconsistent memory modifications, since accessing the checkpoint data is a critical section. In order to avoid this situation, the checkpointing layer creates a replica of the memory blocks

to be dumped before creating the checkpointing thread. This enables concurrency, but it takes time to perform the copying itself. Another option would be to protect the memory to be dumped, allowing only read accesses to it. This can be done by any modern operating system, but it presents several problems that make memory replication a better option:

- There is no standard interface for memory protection, which makes the code non-portable. This disadvantage can be easily overcome, however, since memory protection could be abstracted through the portability layer.
- In most systems the memory protection works at page level. Since CPPC works at variable level, this scheme is too coarse-grained, which would result in many variables being unnecessarily protected when located in the same page than relevant variables, resulting in a potential efficiency loss.
- When a process tries to write to protected memory, it receives a SIGSEGV. In order to avoid execution failure, the library would need to capture and handle this signal while checkpointing takes place. This could mask a legitimate segmentation fault caused by the application code for reasons different than a write to protected memory. This, however, should not happen in correct codes.

Once memory replication is finished, a new thread is created to handle checkpointing file write, performed by the writing layer. CPPC supports sequential checkpointing through the `CPPC/Checkpointer/Multithreaded` configuration parameter. If set to `false`, no threads are created and the checkpointing process uses the original application memory, returning control when the state file has been persistently created in stable storage.

This service is the only one from the checkpointing layer that the façade uses during a normal execution. The rest of them are only relevant during the restart phase.

- Integrity checks

When recovering an execution, each process checks which local checkpoints it has available to build the recovery line. Since CPPC aims to provide fault tolerance, it

cannot overlook the fact that checkpoint files may become corrupt due to a series of facts. Therefore it must perform state file integrity checks.

CPPC provides an abstraction layer on the writing algorithm used for checkpoint file creation. This is due to the desire to provide flexibility and support for different computing platforms. However, this also means that integrity checks must be performed by the writing module that created a certain state file. The implementation of this service, therefore, only checks that the requested file exists, that it is not empty, and delegates all further actions to the writing layer.

- State file reading

This operation is also delegated to the writing layer. The specific writing module used for creating a certain checkpoint file is in charge of recovering its data back for application restart. The only constraint on physical checkpoint file formats is that their first byte must uniquely identify the specific writing module used for its creation. Thus, at this point, the checkpointing layer reads that first byte, and obtains a writing module object by requesting an abstract factory [28] to provide a writer instance identified by that byte. The read operation is then delegated to the obtained writer, that returns the checkpoint in the hierarchical format previously seen in Figure 2.3. This checkpoint object is stored by the checkpointer for servicing future recovery requests by the façade.

- Partial data recovery

This operation is invoked after a checkpoint file has been selected and read into memory. The checkpointing layer is in possession of all checkpoint data, and must return memory blocks to the façade on demand. A partial data recovery operation is invoked to recover the contents of a variable or parameter registration. The checkpointing layer fetches the requested register from the current execution context in the saved hierarchy, and returns the address containing its data.

One of the parameters passed to this function is the size of the register to be recovered, in bytes. This number must be the same as the size of the matching register recovered from the checkpoint file. Note that this number depends on the binary representation for the registered data type in the computing platform used for restart. This mechanism is used as an addition to file integrity checks to further guarantee consistency. The typical errors detected by this technique are inconsis-

tencies between the application code and the file being used for restarting, such as situations where the code has been modified and recompiled after the file was generated.

- Restart completion check

Besides storing variable registrations, call images and subcontexts, an execution context inside a checkpoint file stores whether that file was created while inside said context or not. This function checks two conditions:

- The current execution context in the saved hierarchy is marked as the one where the checkpoint file was created.
- The checkpoint call in the current procedure is the one where the checkpoint file was created. This is done through the unique identifier passed to the checkpoint function, and ensures consistent restart when two or more checkpoint functions are placed in the same procedure scope.

2.3.3. Writing layer

This layer is in charge of actually handling the stable storage of state files. It is abstracted through a pure virtual class [63] that provides four operations: write a checkpoint file, read a checkpoint file, check the integrity of a checkpoint file, and obtain a writer plugin identifier code. This code is unique to each implementation of the virtual class, and guarantees a coherent management of the available writing strategies. CPPC includes an HDF-5[7] [27] writing plugin. In the past, a binary writer was also provided, which intended to provide efficiency in exchange for functionality. It was discontinued, however, since configuring the HDF-5 library to write data in memory format while tagging it to support data conversions when restarting the application, if necessary, resulted in an equally efficient operation.

New writing plugins can be easily implemented and integrated into the library. A writer abstract factory provides a function for writing plugins to register themselves when the application starts. The factory then manages a hash table indexed by writer codes, and containing a constructor function for associated writer objects.

[7] HDF-5 is a hierarchical data format and associated library for the portable transfer of graphical and numerical data between computers.

Plugins are dynamically registered taking advantage of the initialization of constant variables [6]. The factory provides a method for obtaining a writer through its writer code. Thus, the checkpointing layer may perform requests for writer instances in two different ways, depending on the execution state:

- If a state file is being written, the writer code is located through the `CPPC/Writer/Type` configuration parameter.
- If a state file is being read, the writer code is obtained by reading the first byte of the checkpoint file used for restart.

Again, the only constraint for the physical format of generated state files is that their first byte must contain the writer code of their creator. Otherwise, the writing strategy must only guarantee that it is able to consistently recover its written data.

Regarding writing options and functionalities, these are completely dependent on the specific writing module being used. Each particular writing module can use its own namespace to accept configuration parameters (e.g. `CPPC/Writer/HDF-5` for the HDF-5 writer).

2.3.4. Portability layer

A portable tool for checkpointing parallel applications must deal with non-standard interfaces that may risk portability of the library between different hardware/software platforms. For instance, operating systems provide different interfaces for accessing the filesystem. CPPC abstracts certain parts of its functionality through a system of template classes [63] that receive a *policy* [6], or implementation class in which they delegate all their behavior.

The portability layer is static, as oppossed to dynamic ones such as the writing layer. This means that the actual implementation for the abstracted interface is selected at compile time. A recompilation is needed in order to change it later. Since this layer tries to achieve portability, and not extra functionalities, there is no need to make it dynamic, which would in turn diminish efficiency (by forcing the use of virtual functions [63]).

- Filesystem

The problem of portably accessing the filesystem has been mentioned several times. To workaround it, CPPC defines two interfaces that must be implemented for each operating system to be supported by the library. The first interface is the filesystem manager itself, providing operations for creation and removal of folders, removal of regular files (there are standard means for creating regular files in C++), test the existence and size of files, and for opening directory streams. Here the second interface comes into play: the directory stream itself. It is a pure virtual class that must provide a single method to iterate through the directory's contents. Both interfaces provide the necessary abstractions for CPPC operations to be portably implemented in the above layers.

A POSIX[8] implementation of these interfaces is provided with CPPC. It may be used for compiling the library in operating systems conforming to the standard. However, the implementation of these interfaces is quite simple for any given operating system. Currently, a proposal of the Boost.Filesystem library[9], which provides portable facilities to query and manipulate paths, files and directories, has been accepted by the C++ Standards Committee for inclusion in the C++ Technical Report 2. If finally included into the language, this would eliminate the need for this abstraction layer. The inclusion of a Boost.Filesystem dependency in CPPC was also considered, but ultimately discarded since the Boost libraries are not commonplace.

- XML manipulation

This simple abstraction interface decouples CPPC from the specific XML parsing solution being used. It only contains two relevant functions: one for indicating the file to be parsed and another one for recovering the value of a given tag.

An implementation of this interface using the Xerces-C [72] library, a portable widespread parsing facility, is included with CPPC.

[8]POSIX (Portable Operating System Interface) is the collective name of a family of related standards specified by the IEEE to define the API, along with shell and utilities interfaces for software compatible with variants of the Unix operating system, although the standard can apply to any operating system.

[9]The Boost C++ libraries are a collection of peer-reviewed, open source libraries that extend the functionality of C++. They cover a wide range of application domains, ranging from general-purpose libraries (e.g. smart pointers implementations) to OS abstractions like Boost.FileSystem. The Boost libraries homepage can be found at `http://www.boost.org`.

- Communications system

While CPPC was designed to work with MPI applications, the portability of the strategies used for rollback-recovery allows it to be independent from the communications system. Thus, it is enough to abstract communication-related operations through an interface to achieve independence. Operations included in this interface are: global synchronizations (barriers); obtaining a unique rank for each process executing the application; obtaining the total number of processes executing the application; and performing reduction operations on the minimum and maximum of a basic data type. Note that if the communications system does not support reduction operations, these might be implemented using global or point-to-point communications.

Two different implementations of the communications interface are provided: an MPI-based one, and another one for the execution of sequential applications, called "no-communicator", that makes a trivial implementation of these operations in a single-process environment.

2.3.5. Utility layer

This layer includes reusable functionalities useful for implementing writers, communication modules, etc.

- Compression system

Under certain circumstances, the user may need to apply compression to the state files generated by CPPC. Originally, file compression was centralized by the checkpointing layer. However, the abstraction between this and the writing layer generated troublesome situations. For example, compressing a file involved writing it to disk uncompressed through the writing layer, reopening it at the checkpointing layer, compressing the data and writing the file again. This becomes extremely inconvenient when stable storage is remotely provided. Due to this, the system became an independent entity as part of the utility layer. Each writing plugin may decide how to use (or not use) the compression algorithms included with CPPC. This enables the use of ad-hoc compression systems when using complex data storage libraries for checkpoint file storage, like HDF-5, which provides its own compression capabilities. The obvious disadvantage is that decentralizing this operation increases

the complexity of the writing modules, which have to include the compression code themselves.

The structure of the compression system is a replica of the writing layer. Compression algorithms must implement a common interface, which provides methods for data compression and decompression, and register themselves with an abstract factory when the execution starts. Each plugin implementation has its own unique identifier code, so that the factory knows which instance to return. The `CPPC/Compressor/Type` configuration parameter defines which implementation of the compressor to use. Writing plugins using the compression system must somehow store the code for the used compressor into the file, in a similar way to the writer code, in order to enable decompression upon restart.

The impact of compressing files on checkpointing overhead is analyzed in Chapter 4.

- Configuration manager

This subsystem was introduced when covering the `CPPC_Init_configuration()` function in the façade, in Section 2.3.1. Its purpose is to encapsulate configuration sources and priorities. It consists of a singleton class with two public methods: one to initialize the manager, which receives the application command-line parameters and initializes and reads CPPC's configuration file; and another one for queries regarding parameter values.

Since the manager is heavily used by all the layers in the model, it uses a parameter cache to improve the performance of the query operations.

- Cyclic redundancy checks

This system includes a reusable class implementing a common CRC-32 algorithm. It may be used by writing plugins to implement their integrity checks.

2.4. Summary

This chapter describes one of the two fundamental parts of this work: the checkpointing library for message-passing parallel applications. The design and imple-

mentation of the library have been described using a top-down approach, beginning at the application code and descending through the controller, used for decoupling the model from the programming language being used, and finally through the model layers of CPPC: façade, checkpointing layer and writing layer. Also, the portability layer has been described, being specially important for providing abstractions for certain OS-dependent operations (filesystem access, communications, etc.). Finally, the utility layer, which combines useful functionalities offered to all the model layers, was described.

As seen throughout the chapter, the modifications that need to be performed on the application's code in order to achieve integration with CPPC are far from trivial. To automate this task, a source-to-source compiler is used, as described in the following chapter.

Chapter 3

CPPC Compiler

The integration of the CPPC library with a parallel application requires modifications to be performed to the original code. Their specifics were detailed in the previous chapter. If manually performed, the process would place an undesirable burden upon the users, who would have to manually perform complex analyses such as detecting live variables and safe points. Besides inserting CPPC library calls into the appropriate places, users would also need to insert the control flow code that enables the recovery process. In order to free users from these tasks, fulfilling the transparency goal in CPPC, a source-to-source compiler has been implemented that automates all necessary analyses and transformations to parallel or sequential applications.

This chapter covers all the analyses implemented into the CPPC compiler, as well as some of its most relevant internal implementation details.

3.1. Compiler overview

The CPPC compiler is built on the Cetus compiler infrastructure [37]. It is written in Java, which makes its code inherently portable. Although Cetus was originally designed to support C codes, we have extended it to allow for parsing Fortran 77 codes as well. The specifics of this extension are detailed in Section 3.4. The implementation uses the same basic intermediate representation language (IRL) for both C and Fortran 77 codes, hence allowing the same analysis and transformation code to be used with applications written in both languages.

Some of the analyses performed by the compiler require knowledge about procedure semantics (e.g. which procedure initializes the parallel system). We call these transformations *semantic-directed*. In order to provide the required semantic knowledge to the compiler, CPPC uses a catalog of procedures and its semantic behaviors. An example of the contents of this semantic catalog is shown in Figure 3.1, that exemplifies information for the POSIX `fopen` and `open` functions. Both implement the `CPPC/IO/Open` role (i.e. they open a file I/O stream). Both functions accept two parameters: a path string and a file opening mode, and both return a file descriptor. Each procedure in the semantic catalog may implement any number of semantic roles. Each semantic role indicates that the function has a particular semantic behavior. The implementation of a role by a function may have associated attributes indicating the specifics of the semantic behavior. For instance, the `DescriptorType` attribute for the `fopen/open` case indicates the type of the returned descriptor. It is internally used by the filesystem module in the portability layer of the CPPC library to select a specific file stream manipulation routine, depending on the returned descriptor type.

The catalog also contains data flow information. Procedure parameters are categorized into `input`, `output` and `input-output`, depending on whether they are read, written, or read and written by the procedure. These tags allow the compiler to perform optimized data flow analyses for live variable detection (see Section 3.2.5). However, data flow information is not necessary for proper operation. If missing, the compiler will take the conservative approach of considering all function parameters to be of input type, which implies that their values need to be recovered before the procedure is called.

The semantic catalog is a portable and extensible solution for providing semantic information, and enables transformations to work with different programming interfaces for a given subsystem. Supporting PVM communications, for instance, only requires the addition of the corresponding semantic information. Note that the semantic catalog is not created or modified by CPPC users, but provided together with the compiler distribution. In the current CPPC distribution, the catalog includes information about Fortran 77 built-in functions; common UNIX functions; and the MPI standard, in both C and Fortran 77 versions. This is enough to support most HPC MPI application in *NIX environments. Table 3.1 summarizes all existing semantic roles and their attributes.

```
<function name="fopen">
  <input parameters="1,2"/>
  <semantics>
    <semantic role="CPPC/IO/Open">
      <attribute name="FileDescriptor"
        value="return"/>
      <attribute name="Path" value="1"/>
      <attribute name="DescriptorType"
        value="CPPC_UNIX_FILE"/>
    </semantic>
  </semantics>
</function>

<function name="open">
  <input parameters="1,2"/>
  <semantics>
    <semantic role="CPPC/IO/Open">
      <attribute name="FileDescriptor"
        value="return"/>
      <attribute name="Path" value="1"/>
      <attribute name="DescriptorType"
        value="CPPC_UNIX_FD"/>
    </semantic>
  </semantics>
</function>
```

Figure 3.1: Semantic information for the `fopen` and `open` functions

Table 3.1: Semantic roles used by the CPPC compiler

Role	Semantic	Attributes
`CPPC/Comm/Initializer`	Initialization of the parallel subsystem	None
`CPPC/Nonportable`	Non-portable function call	None
`CPPC/IO/Open`	Opening of a file stream	• `FileDescriptor`: How is the descriptor returned. • `Path`: Path parameter. • `DescriptorType`: Type of the file descriptor.
`CPPC/IO/Close`	Closing of a file stream	• `FileDescriptor`: Descriptor parameter.
`CPPC/Comm/Send`	Message send	• `Blocking`: true/false. • `Type`: P2P/Collective. • `Peer`: Peer rank parameter. • `Tag`: Tag parameter. • `Communicator`: Communicator parameter. • `Request`: Request parameter, for non-blocking communications only.
`CPPC/Comm/Recv`	Message receive	Same as `CPPC/Comm/Send`.
`CPPC/Comm/Wait`	Communication wait	• `Blocking`: true/false. • `Type`: P2P/Collective. • `Request`: Request parameter, for non-blocking waits only.
`CPPC/Comm/Ranker`	Obtention of the rank of a process in a communicator	• `Rank`: How is the rank returned.
`CPPC/Comm/Sizer`	Obtention of the size of a communicator	• `Size`: How is the size returned.

3.2. Analyses and transformations

The CPPC compiler is a multi-pass compiler. These kind of compilers process the source code multiple times. Each pass takes the results of the previous one as its input, and creates an intermediate output. The code is improved pass by pass, until the final code is emitted. The first of these passes is the parsing of the source code, that builds the Cetus AST[1]. Once this is done, the compiler begins the sequential application of the set of passes that perform the core of the CPPC analyses and transformations for checkpointing instrumentation. When information must be transferred between passes, the compiler uses different forms of AST annotation. The following list of compiler passes is exhaustive, and described in the order the compiler applies them:

3.2.1. CPPC initialization and finalization routines

The `CPPC_Init_configuration()` call initializes the CPPC configuration, while `CPPC_Init_state()` marks the point for state initialization, creating all necessary memory structures, such as lists of registered variables, execution contexts, etc. When resuming the execution of a parallel application, interprocess communications are performed in order to find the recovery line. Therefore, the state must be initialized after the communications system (e.g. after `MPI_Init()` for MPI applications). The insertion of `CPPC_Init_configuration()` is straightforward, and performed at the first line of the application's entry procedure. The transformation for inserting `CPPC_Init_state()` is semantic-directed. The compiler looks for a procedure implementing the `CPPC/Comm/Initializer` role. If found, it inserts the state initialization right after it. Otherwise, the application is presumed to be sequential or to use a communication system not needing initialization, and the call is inserted after `CPPC_Init_configuration()`.

The finalization function, `CPPC_Shutdown()`, is inserted at exit points in the application code. It ensures that ongoing checkpoint operations are correctly finished before the application ends.

[1] An Abstract Syntax Tree (AST) is a tree representation of the syntax of some source code. Each node of the tree denotes a construct occurring in the source code. The tree is *abstract* in the sense that it may not represent some constructs that appear in the original source (e.g. grouping parenthesis, which are not needed since the grouping of operands is implicit in the tree structure).

```
CPPC JUMP LABEL
CPPC_Create_call_image( "MPI_Comm_split", line_number );
CPPC_Register_parameter( &color, 1, CPPC_INT, "color", CPPC_STATIC );
CPPC_Register_parameter( &key, 1, CPPC_INT, "key", CPPC_STATIC );
CPPC_Commit_call_image();
MPI_Comm_split( comm, color, key,comm_out );
CONDITIONAL JUMP TO NEXT REC
```

Figure 3.2: Example of a non-portable procedure call transformation

3.2.2. Procedure calls with non-portable outcome

The CPPC library recovers non-portable state by means of the re-execution of the code responsible for creating such state in the original run. The `CPPC/Nonportable` semantic role is used for identifying procedures that create or modify non-portable state. Upon discovery of a non-portable call, the compiler performs the transformation depicted in Figure 3.2 for an `MPI_Comm_split()` call, where the CPPC instrumentation added by the compiler is marked in bold. The block begins with a label annotation which marks the jump destination from the previous REC, and ends with a conditional jump annotation, to mark the point where a jump to the next REC must be performed. Both annotations are objects in the `cppc.compiler.ast` package. The compiler also inserts a register for each `input` or `input-output` parameter passed to the call that is a portable object. For more information on the runtime behavior of non-portable calls and their instrumentation refer to Section 2.1.3.

3.2.3. Open files

In order to identify file opening calls, a semantic role `CPPC/IO/Open` is used. This role accepts the following attributes:

- `FileDescriptor`: A numeric value, or the special value `return`. It tells the compiler which of the procedure parameters holds the file descriptor after the call, or if it is available as the returned value.
- `Path`: A numeric value indicating the parameter that contains the file path.
- `DescriptorType`: A special constant value that must be defined and accepted by the filesystem module in the portability layer. Current accepted values

```
CPPC JUMP LABEL
fd = open( path, mode );
CPPC_Register_descriptor( desc_id, &fd, CPPC_UNIX_FD, path );
CONDITIONAL JUMP TO NEXT REC
```

Figure 3.3: Pseudocode for a file opening transformation

```
CPPC_Unregister_descriptor( &fd );
close( fd );
```

Figure 3.4: Pseudocode for a file closing transformation

are `CPPC_UNIX_FD`, indicating that this is an integer like the ones returned by `open`, and `CPPC_UNIX_FILE`, that identifies `FILE *` descriptors as the ones returned by `fopen`.

The compiler uses this information to perform the transformation depicted in Figure 3.3, where instrumentation code is emphasized in bold. The `desc_id` parameter passed to the descriptor registration call is a unique identifier for this descriptor, automatically assigned by the compiler. When the application is recovered, `CPPC_Register_descriptor()` ensures correct repositioning of the file stream pointer.

File closing calls are identified by the `CPPC/IO/Close` role, parameterized only with the `FileDescriptor` attribute defined as above. The transformation performed by the compiler is shown in Figure 3.4. Note that this block is not enclosed by control flow annotations, because it does not need to be re-executed for proper state recovery.

3.2.4. Conversion to CPPC statements

Some of the transformations to be performed from this point require statements to be intensively annotated. However, the Cetus class for representing a statement does not include any means for this. Thus, a `cppc.compiler.ast.CppcStatement` class was created, through the specialization of the original `cetus.hir.Statement`. It acts as a proxy for a contained regular statement, and adds some annotation structures, such as data flow information (i.e. variables that are read or written by the statement), information about whether a particular statement is a safe point,

etc. This transformation makes a pass over the entire application code substituting regular Cetus statements for this CPPC version, enabling information sharing amongst subsequent transformations.

3.2.5. Data flow analysis

In order to identify the variables needed upon application restart, the compiler performs a live variable analysis. This is a somehow complementary approach to memory exclusion techniques used in sequential checkpointers to reduce the amount of memory stored, such as the one proposed in [48].

A variable x is said to be *live* at a given statement s in a program if there is a control flow path from s to a use of x that contains no definition of x prior to its use. The set LV_{in} of live variables at a statement s can be calculated using the following expression:

$$LV_{in}(s) = (LV_{out}(s) - DEF(s)) \cup USE(s)$$

where $LV_{out}(s)$ is the set of live variables after executing statement s, and $USE(s)$ and $DEF(s)$ are the sets of variables used and defined by s, respectively. This analysis, that takes into account interprocedural data flow, is performed backwards, being $LV_{out}(s_{end}) = \emptyset$, and s_{end} the last statement of the code.

Before each checkpoint statement c_i, the compiler inserts annotations to register the variables that must be stored in the checkpoint file, which are those contained in the set $LV_{in}(c_i)$. The data type for the register is automatically determined by checking the variable definition. Variables registered or defined at previous checkpoints are not registered again. Also, before each checkpoint c_i, the compiler inserts "unregister" annotations for the variables in the set $LV_{in}(c_{i-1}) - LV_{in}(c_i)$, the set of variables that are no longer relevant.

Checkpoints can be placed inside any given procedure. For a checkpoint statement c_i, let us define:

$$B_{c_i} = \{s_1 < s_2 < \ldots < s_{end}\}$$

as the ordered set of statements contained in all control flow paths from c_i (excluding c_i) and up to the last statement of the program code, where the $<$ operator indicates the precedence relationship between statements. Note that, if a checkpoint is placed inside a procedure f, not all statements in the set B_{c_i} will be inside f. Let us

denote by $B^f_{c_i} = \{s_1 < \ldots < s_n\}$ the ordered set of statements contained inside f, and $L^f_{c_i} = B_{c_i} - B^f_{c_i}$ the ordered set of statements left to be analyzed outside f.

The interprocedural analysis and register insertion is performed according to the following algorithm:

- For a checkpoint statement c_i contained in a procedure f, the live variable analysis is performed for the set $B^f_{c_i}$, and registers for locally live variables are inserted before c_i.
- For the set $L^f_{c_i}$, containing the statements that are left unanalyzed in the previous step, let us consider g to be the procedure containing the statement s_{n+1}. The statement executed immediately before s_{n+1} must be a call to f. Note that $L^f_{c_i} = B^g_{c_i} \sqcup L^g_{c_i}$.

  The live variable analysis is performed for the set $B^g_{c_i}$, and registers for locally live variables are inserted before the call to f.
- The process is repeated for the statements contained in the ordered set $L^g_{c_i}$.

This algorithm ensures that, upon application restart, all variables will be defined before being used, and thus the portable state of the application will be correctly recovered.

Proof of Correctness: Let us consider a variable v which appears in the statements contained in B_{c_i}, and let $s_v \in B_{c_i}$ be the statement where it first appears.

There are three different cases to analyze: v can either be an `input`, an `output`, or an `input-output` variable for s_v. Using the definition of the live variable analysis:

- `input`: $v \in USE(s_v) \wedge v \notin DEF(s_v)$

  $$USE(s_v) \subset LV_{in}(s_v) \Rightarrow v \in LV_{in}(s_v)$$

  Since s_v was the first appearance of v in B_{c_i}, there is no previous statement which defines its value, meaning that v belongs to the live variable set for every statement before s_v:

  $$\nexists i / (v \in DEF(s_i)) \wedge (s_i < s_v) \Rightarrow$$

  $$v \in LV_{in}(s_j), \forall j / s_j < s_v$$

In particular, since $c_i < s_v$: $\mathbf{v \in LV_{in}(c_i)}$.

A register will be inserted before s_v when analyzing its containing procedure. This register will generate v's value when restarting the application.

- `output`: $v \notin USE(s_v) \wedge v \in DEF(s_v)$

  In this case, it is guaranteed that $v \notin LV_{in}(s_v)$.

  Since s_v was the first appearance of v in B_{c_i}, there is no previous statement which uses its value, meaning that v does not belong to the live variable set for every statement before s_v:

$$\nexists i / (v \in USE(s_i)) \wedge (s_i < s_v) \Rightarrow$$

$$v \notin LV_{in}(s_j), \forall j / s_j < s_v$$

  In particular, since $c_i < s_v$: $\mathbf{v \notin LV_{in}(c_i)}$.

  No register will be inserted; v's value will be generated upon reaching s_v, therefore defining it.

- `input-output`: $v \in USE(s_v) \wedge v \in DEF(s_v)$

  This case is similar to the input one.

■

Proof of Termination: The algorithm terminates if all statements contained into B_{c_i} are analyzed. B_{c_i} is a finite set, since its maximum number of elements is equal to the total number of statements in the application being analyzed. The evolution of the cardinality of the unanalyzed set of statements is as follows:

- In the first phase of the algorithm (the analysis of procedure f), the statements in $B^f_{c_i}$ are analyzed. Either $s_1 \in f$ and $\#L^f_{c_i} < \#B_{c_i}$, or c_i is the last statement in f and $\#L^f_{c_i} = \#B_{c_i}$.

- In each of the subsequent phases, the set of unanalyzed statements can be written as:

$$L^{p_n}_{c_i} = \{s^{p_n}_1, \ldots, s^{p_n}_m\} = B_{c_i} - (\bigsqcup_{j=1}^{n} B^{p_j}_{c_i})$$

where each p_j is the procedure being analyzed in phase j. In phase n, the algorithm analyzes the procedure containing the statement $s_1^{p_n}$. Therefore, at least one statement is analyzed, and $\#L_{c_i}^{p_{n+1}} < \#L_{c_i}^{p_n}$.

Since all the statements must be contained in a procedure in order to be in the initial B_{c_i} and this is a finite set, the termination of the algorithm in a finite number of steps is guaranteed.

■

The compiler does not currently perform optimal bounds checks for pointer and array variables. This means that some arrays and pointers are registered in a conservative way: they are entirely stored if they are used at any point in the re-executed code.

When dealing with calls to precompiled procedures located in external libraries, the default behavior is to assume all parameters to be of input type. Therefore, registration calls will be inserted for the previously unrecovered variables contained in the set $LV_{in}(s_p)$, being s_p the analyzed procedure call. To avoid this default behavior, the CPPC compiler can use data flow information available in the semantic catalog, as explained in Section 3.1.

3.2.6. Communication analysis

In order to automatically find regions in the code were neither in-transit nor inconsistent communications exist, that is, safe points, the compiler must analyze communication statements and match sends to their respective receives. The approach the compiler uses is similar to a static simulation of the execution. Two communications are considered to match if two conditions hold:

1. Their tags are the same or the receive uses `MPI_ANY_TAG`.
2. Their sets of sources/destinations are the same: if a process i sends a message to a process j, then j must execute the corresponding receive statement using i as source.

In order to statically compare tags and source/destination pairs, the compiler performs constant propagation and folding to determine their literal values during

the execution. Since propagating and folding all application constants may prove to be as computationally intensive as the actual execution, the compiler restricts this operation to the set of variables that are involved in the calculations of parameters affecting communications. The set of *communication-relevant* variables is recursively calculated as:

1. Variables directly used in tags or source/destination parameters.
2. Variables on which a communication-relevant variable depends.

In order to optimize this process, only statements that modify communication-relevant variables are analyzed for constant folding, since any other does not affect communications. The data flow information is obtained from the annotations introduced by the previous compilation pass. Some communication-relevant variables are multivalued, that is, they potentially have a different value for each process. Thus, the compiler needs to know how many processes are involved in the execution of the code to perform the constant folding. For instance, many applications are written in a scalable fashion, using code that dynamically calculates neighbor processes in the commmunication topology. These calculations are affected by the total number of processes involved in the parallel execution. The number of processes is provided to the compiler via command-line parameters. Then, using the `CPPC/Comm/Ranker` and `CPPC/Comm/Sizer` semantic roles, the compiler is able to find values for the variables that contain the number of processes and the rank of a process. Note that the latter is multivalued (e.g. $\{0, \ldots, N-1\}$ if there are N processes in the execution). Expressions derived from multivalues are, in the general case, multivalued themselves. The constant folding and propagation is performed together with the communication matching in the same compilation pass. The compiler is able to fold array expressions and correctly calculate array values statically, as long as they depend entirely on values known at compile time.

For keeping track of the communications status, the compiler uses a buffer object, which starts out empty. The analysis begins at the application's entry point. Statements that are neither control flow- nor communication-related are ignored. Each time the compiler finds a new communication, it first tries to match it with existing ones in the buffer. If a match is not found, the communication is added to the buffer and the analysis continues. If a match is found, both statements are considered linked and removed from the buffer, except when matching non-blocking sends and receives, in which case they remain in the buffer in an unwaited status until a matching wait is found.

A statement in the application code will be considered a safe point if, and only if, the buffer is completely empty when the analysis reaches that statement. An empty buffer implies that no pending communications have been issued, and therefore it is impossible for an in-transit or inconsistent message to exist at that point.

The following subsections deal with particular aspects of the communications analysis, such as analyzing conditional expressions, procedure calls, collective communications and the limitations of this approach.

- Conditional statements

Conditional statements need special treatment, since their execution order is nondeterministic. Upon reaching a conditional, SPMD processes potentially divide into two groups: those executing the true clause, and those executing the false one. The compiler uses *linked buffers* to analyze conditional statements. The difference with a regular buffer is that a linked buffer B_l is associated to another one B_o in a way that, for a statement to be considered a safe point, not only the B_l has to be empty, but B_o must also be consistent with the safe point condition. Note that this may be generalized as a chain of linked buffers which check if they are themselves empty and delegate the final decision on their link until one is not empty or the root one is reached, much like in a chain of responsibility pattern [28].

The compiler creates an empty buffer B_t linked to the old one B_o, and uses it to analyze the true clause. Found communications are added to B_t, and matched as usual, with an important difference: each one is marked as being executed only if the expression controlling the conditional statement execution holds. After the true clause is analyzed, the compiler proceeds in the same way with the false one using a different buffer B_f also linked to B_o. This time, each communication is marked as being executed only if the expression controlling the conditional does not hold.

When both branches have been fully analyzed, the compiler has three different buffers, B_t, B_f and B_o, that need to be merged into a single one. First, *redundant* communications are removed from B_t and B_f. Although both conditional paths execute a different set of statements, equivalent communications may exist. These are communications that are executed in different statements in the code but matched to a single one outside the conditional. If these redundant communications were not identified, the algorithm would yield incorrect results, since only one of the equivalent communications would be correctly matched. Thus, two communication

statements are considered equivalent if they both match the same set of statements according to their source/destination pair and tag, and if both are issued under incompatible control expressions (e.g. on different branches of the same conditional statement). Redundant communications are substituted by a single representative one, that will be matched at the same point in the code where each of the redundant ones would during execution. This enables consistent discovery of safe points in the presence of redundant communication statements.

The next step consists in merging all three buffers into a single one that represents the communications state after executing the conditional. The process of merging two buffers is a binary operation. The left hand side buffer (B_{lhs}) represents the communications state before executing a block of code. The right hand side buffer (B_{rhs}) represents the communications issued by the execution of that block of code. The communications in B_{rhs} are orderly injected into B_{lhs}. Non matching communications are added to B_{lhs}. Matches are dealt with as previously explained. When the process ends, the resulting buffer represents the communications state of a process that, starting in a state represented by B_{lhs}, executes the block of code issuing communications in B_{rhs}. Note that this process implies a deferral of the matches for the statements in the new buffer. Since there is no correct partial order for the statements in conditional branches, both are independently analyzed and then mixed before being injected into the buffer representing the old state.

First, B_t and B_f are merged together. The result is merged with B_o. The result of this second merge is used for the analysis of subsequent communications. The reason for merging the buffers obtained from analyzing the conditional first is that often communications in different conditional branches match between themselves.

Note that some conditional control expressions may be calculated during the constant folding analysis. In these cases, the compiler analyzes only the appropriate conditional branch. The compiler is also able to detect situations in which the control expression is multivalued, and deal with them accordingly.

- Procedure calls

When the compiler finds a procedure call, it stops the ongoing analysis and moves to analyze the code of the called procedure, using the same communications buffer. However, communications issued inside the procedure are also cached separately for optimization purposes. If the procedure does not modify any communication-

relevant variable, cached results can be reused when a new call to the same procedure is found, without analyzing the procedure code, but just symbolically analyzing their tags and source/destination parameters again.

The compiler employs this optimization whenever possible, thus avoiding the repeated analysis of procedures each time a call to them is found. If the procedure modifies any communication-relevant variable, then its code is analyzed each time a call is found, since these modifications have to be tracked and applied to guarantee correct constant folding.

- Collective communications

In SPMD applications, the usual way of coding collective communications is to make all involved processes execute a single, non conditional line of code. Before and after the execution of that line the collective communication does not affect the consistency of communications. However, in the general case, a code may contain a collective communication spread across several conditional paths. In this case, all its parts must be detected as redundant and the compiler must ensure that no checkpoints are inserted such that some processes take its local one before the collective while some others do so after it.

- Limitations

Some parallel applications present nondeterministic or irregular communication patterns (those depending on runtime input data, where tags and/or source/destination parameters are derived from the input data and cannot be statically determined). In these situations, the compiler must apply either a conservative approach or heuristics in order to detect safe points. A conservative approach involves matching communications to their latest possible match in the code, although this can lead to the compiler not finding safe points at points selected by the checkpoint insertion analysis described in the next section. The heuristic currently in use is to consider that two communications match if no matching incompatibility between their tags and source/destination pairs is found.

3.2.7. Checkpoint insertion

The compiler locates sections of the code that take a long time to execute, where checkpoints are needed to guarantee execution progress in the presence of failures. Since computation time cannot be accurately predicted at compile time without knowledge of the computing platforms and input data, heuristics are used. The compiler discards any code location that is not inside a loop, and ranks all loop nests in the code using computational metrics. Currently, a metric derived from both the number of statements executed inside the loop and the number of variables accessed in them is used. A loop might have many statements but accessing mostly constants. These are usually not good candidates for checkpointing, since they involve initializing variables and their execution does not last long. In this case, considering the number of variable accesses in the loop reduces the cost associated to such loops, thus making them less prone to be checkpointed.

Let l be a loop in L, the loop population of the application P, and s and a two functions that give the number of statements and variable accesses in a given block of code, respectively. Let us define $S(l) = s(l)/s(P)$ and $A(l) = a(l)/a(P)$ the total proportion of statements and accesses, in that order, that exist inside a given loop l. The heuristic complexity value associated to each loop l is calculated as:

$$h(l) = -log(S(l) \cdot A(l)) \tag{3.1}$$

Eq. (3.1) multiplies $S(l)$ and $A(l)$ to ensure that the product is bigger for loops that are significant for both metrics. It applies a logarithm to make variations smoother. Finally, it takes the negative of the value to make $h(l)$ strictly positive. Thus, a lower value implies that more computing time is estimated for that loop. In order to select the best candidates for checkpoint insertion, the compiler ranks loops in L attending to $h(l)$ and applies thresholding methods [57]. After exploring several possibilities, a two-step method has been adopted. First, a "shape-based" approach is used to select the subset of the time-consuming loop nests in the application. The second step improves this selection by means of a "cluster-based" technique. Figures 3.5 and 3.6 show the proposed method applied to the BT application of the NAS Parallel Benchmarks [3].

In the first step, using the so-called "triangle method", loops in L are divided into two classes: time-consuming loops and negligible loops. Conceptually, the triangle method consists in: (a) drawing a line between the first and last values of $h(l)$; (b) calculating the perpendicular distance $d(l)$ to this line for all $l \in L$; and (c)

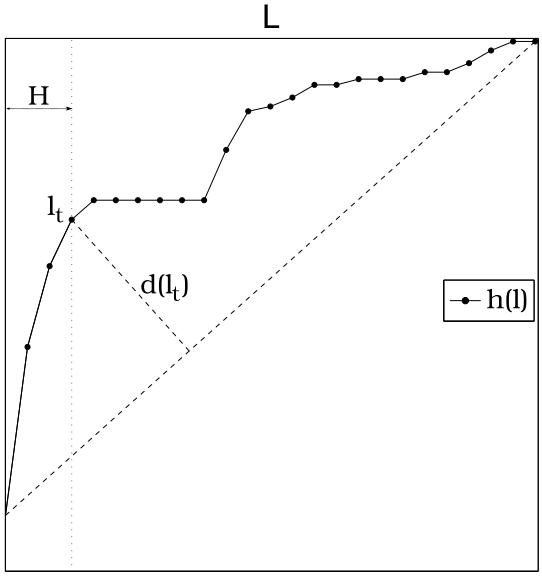


Figure 3.5: First step: shape-based threshold

calculating a threshold value l_t such that $d(l_t) = max\{d(l)\}$. This first step selects a subset of the loop population, referred to as H, as time-consuming loops candidates for checkpoint insertion, as shown in Figure 3.5. The triangle method was originally developed in the context of image processing [73], and appears to be especially effective when there is a narrow peak in the histogram, which is often the case for the proposed $h(l)$ function in real applications.

However, this first threshold selects more loops than would be desirable for checkpoint insertion. The second step of the proposed algorithm refines this selection using a cluster-based thresholding algorithm. Loops in H with similar associated costs are grouped into clusters, C_i, built by selecting the local maximums of the second derivative of $h(l)$ as partitioning limits. Since $h(l)$ is monotonically increasing, local maximums in $h''(l)$ represent inflection points at which $h(l)$ begins to change more smoothly. For an application with k clusters, the method calculates a threshold value t such that:

$$\sum_{i=0}^{t} h(l_{i+1}^0) - h(l_i^0) > \sum_{i=t+1}^{k-1} h(l_{i+1}^0) - h(l_i^0) \tag{3.2}$$

where l_i^0 denotes the first loop in C_i. This method selects for checkpoint insertion all the loops inside the clusters C_i such that $i \leq t$. In the example shown in Figure 3.6, loops in subset H are grouped into two clusters, delimited by the single maximum

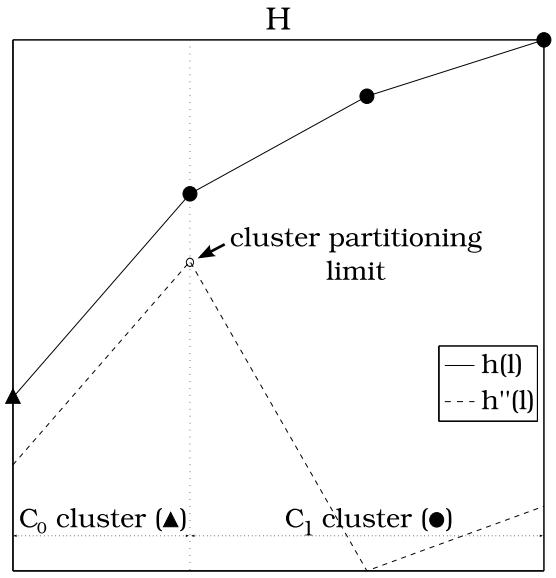


Figure 3.6: Second step: cluster-based threshold

in $h''(l)$. Applying Eq. (3.2), the compiler selects cluster C_0 for checkpoint insertion, which contains only one loop (which corresponds to the main computational loop in BT).

Once the loops in which checkpoints are to be inserted are identified, the compiler uses the communication analysis described in the last subsection to insert a checkpoint at the first available safe point in each selected loop nest. Experimental results are detailed in Chapter 4.

- Manual checkpoint insertion

Instead of, or in addition to, automatically placed checkpoints, the CPPC compiler allows for manual insertion of checkpoints through two different user-inserted directives:

- `#pragma cppc checkpoint`: This directive orders the compiler to place a checkpoint call in the exact position where it is inserted.
- `#pragma cppc checkpoint loop`: This directive is placed before a loop, and tells the compiler to insert a checkpoint at the first available safe point inside it.

```
DO I=A,B,STEP                    CPPC_INIT_IT = A
   !$CPPC CHECKPOINT             CPPC JUMP LABEL
   ...                           CPPC_REGISTER( CPPC_INIT_IT, 1,
END DO                             CPPC_INT, "CPPC_INIT_IT" );
                                 DO I=CPPC_INIT_IT,B,STEP
                                    CPPC_INIT_IT=I
                                    !$CPPC CHECKPOINT
                                    ...
                                 END DO
```

Figure 3.7: Modifications to a checkpointed Fortran 77 loop

Cetus represents compilation directives through its `cetus.hir.Annotation` class, that encapsulates a `String` object that contains the annotation text. Instead, the CPPC compiler defines a `cppc.compiler.ast` Java package, defining several `Annotation` subclasses that are internally used for inter-pass communications. This achieves strong typing of the CPPC annotations, allowing for the compiler to uniformly find and process them using object-oriented techniques such as method overloading. An initial pass translates user-inserted checkpoint directives to the internal CPPC representation.

The Fortran 77 syntax for inserting manual directives is `!$CPPC CHECKPOINT` and `!$CPPC CHECKPOINT LOOP`.

3.2.8. Language-specific transformations

Some language-dependent modifications are necessary in order for the restart protocol to consistently work. Particularly, restarting a Fortran 77 execution from a checkpoint inside a loop needs the loop bounds to be changed in order for the execution to be resumed from the correct loop iteration. Fortran 77 does not accept loop indexes to be dynamically changed once the control flow has entered the loop. Instead, the set of values for the iteration variable is constructed when entering the loop, and its value is updated at each iteration regardless of internal changes. Therefore, a modification to the loop variable persists only until the next iteration starts, where it is modified again and the modification lost.

Figure 3.7 depicts the changes made to a Fortran 77 loop in order to enable internal checkpointing. During a regular execution, the start value `A` is assigned to a

CPPC-introduced variable called `CPPC_INIT_IT`, which is updated at each iteration of the loop before the checkpoint call. Thus, it is guaranteed that it will be stored in the state file containing the value of the checkpointed iteration. When the application is restarted, this value will be recovered through the variable registration, and the original loop bounds altered, starting the iteration at the correct value.

3.2.9. Code generation

At this point, all required analyses have been performed, and the code has been annotated with objects in the `cppc.compiler.ast` package. This last pass looks for those annotations and replaces them with their corresponding code in the AST. Then, a code generator reads the AST and generates the output in the original programming language.

3.3. Case study

For illustrative purposes, an example of how CPPC inserts checkpointing code into a parallel application is presented. The target application is a simplified C version of a large-scale engineering code that performs a crack growth simulation using the Dual Boundary Element Method (DBEM) [30], further used in Chapter 4 for the experimental evaluation of the CPPC tool. Figure 3.8 shows the original code, which can be divided in five basic sections:

1. Reading of the execution parameters from a configuration file (done by the process with rank 0).

2. Creation of the necessary MPI structures, such as communicators and data types for supporting the communication between processes.

3. Distribution of the execution parameters and data previously read by process 0 among all the application processes.

4. Call to the `frmtrx()` subroutine, responsible for building the system matrix (variable `a`) and the right-hand side vector (variable `b`) of the equation system used for modelling the engineering problem.

```
int main( int argsc, char ** argsv ) {
  // Variable definitions
  MPI_Init( &argsc, &argsv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

  // Host reads parameters
  if( myrank == 0 ) {
    iinput = fopen( "control.dat", "r" );
    fread( iinput, "%s\n", outname );
    fread( iinput, "%d\n", &numstep );
    ...
  }

  // Creation of communication topology
  MPI_Cart_create( MPI_COMM_WORLD, ndims,
    dims, periods, reorder, &cart_comm );
  ...

  // Distribution of parameter data
  MPI_Bcast( &nel, 1, MPI_INTEGER, 0,
    cart_comm );
  MPI_Bcast( &nnp, 1, MPI_INTEGER, 0,
    cart_comm );
  ...

  // Construction of system matrix and rhs
  frmtrx();

  // Solution of equation system
  solve();

  ...
}

void solve() {
  // Variable definitions
  ...
  for( i=0; i < maxiter; i++ ) {
    mpi_dgemv( ml, nl, a, xm, ax, cart_comm );
    mpi_dcopy( ml, b, 1, r, 1, cart_comm );
    mpi_daxpy( ml, -1, ax, 1, r, 1, cart_comm );
    ...
  }
  ...
}
```

Figure 3.8: Case study: original DBEM code

```
int main( int argsc, char ** argsv ) {
  // Variable definitions
  void * cppc_jump_points[4] =
    { &&CPPC_EXEC_1, &&CPPC_EXEC_2, ...}
  int cppc_next_jump_point = 0;
  MPI_Init( &argsc, &argsv );
  CPPC_Init( &argsc, &argsv );
  // Conditional jump to CPPC_EXEC_0
  if( CPPC_Jump_next() ) {
    goto * cppc_jump_points[ jump_index++ ];
  }
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

  // Host reads parameters
CPPC_EXEC_0:
  if( myrank == 0 ) {
    // Conditional jump to CPPC_EXEC_1 (code omitted)
    iinput = fopen( "control.dat", "r" );
CPPC_EXEC_1:
    CPPC_Register_descriptor( 0, iinput,
      CPPC_UNIX_FILE, "control.dat" );
    // Conditional jump to CPPC_EXEC_2 (code omitted)
    fread( iinput, "%s\n", outname );
    ...
  }

  // Creation of communication topology
CPPC_EXEC_2:
  CPPC_Register_parameter( &ndims, ..., CPPC_STATIC );
  dims = CPPC_Register_parameter( dims, ..., CPPC_DYNAMIC );
  periods = CPPC_Register_parameter( periods, ..., CPPC_DYNAMIC );
  CPPC_Register_parameter( &reorder, ..., CPPC_STATIC );
  MPI_Cart_create( MPI_COMM_WORLD, ndims,
    dims, periods, reorder, &cart_comm );
  ...
  // Conditional jump to CPPC_EXEC_3 (code omitted)

  // Distribution of parameter data
  MPI_Bcast( &nel, 1, MPI_INTEGER, 0, cart_comm );
  MPI_Bcast( &nnp, 1, MPI_INTEGER, 0, cart_comm );
  ...

  // Construction of system matrix and rhs
  frmtrx();

  // Solution of equation system
CPPC_EXEC_3:
  CPPC_Context_push( "solve", 0 );
  solve();
  CPPC_Context_pop();
  ...
  CPPC_Shutdown();
}
```

Figure 3.9: Case study: CPPC-instrumented DBEM code

```
void solve() {
   // Variable definitions + CPPC definitions
   // Conditional jump to CPPC_EXEC_4 (code omitted)
   ...
CPPC_EXEC_4:
   CPPC_Register( &i, ..., CPPC_STATIC );
   CPPC_Register( &ml, ..., CPPC_STATIC );
   a = CPPC_Register( a, ..., CPPC_DYNAMIC );
   xm = CPPC_Register( xm, ..., CPPC_DYNAMIC );
   ax = CPPC_Register( ax, ..., CPPC_DYNAMIC );
   b = CPPC_Register( b, ..., CPPC_DYNAMIC );
   ...
   // Conditional jump to CPPC_EXEC_5 (code omitted)
   for( i=0; i < maxiter; i++ ) {
CPPC_EXEC_5:
      CPPC_Do_checkpoint( 0 );
      mpi_dgemv( ml, nl, a, xm, ax, cart_comm );
      mpi_dcopy( ml, b, 1, r, 1, cart_comm );
      mpi_daxpy( ml, -1, ax, 1, r, 1, cart_comm );
      ...
   }
   ...
}
```

Figure 3.10: Case study: CPPC-instrumented DBEM code (cont.)

5. Call to the `solve()` subroutine, containing the computational loop which solves the system.

The resulting code after processing by the CPPC compiler can be seen in Figure 3.9 and Figure 3.10. The compiler inserts new variables which are used to control the program flow on application restart, ensuring that only relevant blocks of code are executed. The applied transformations are the following:

1. The compiler detects that the `MPI_Init()` call implements the `CPPC/Comm/Initializer` role. Therefore, the `CPPC_Init()` call is placed right after it. The finalization function is inserted at the end of the code.

2. The compiler identifies the `fopen()` function as implementor of the `CPPC/IO/Open` role and inserts the C version of the file handling function, `CPPC_Register_descriptor()`.

3. The compiler finds that the `MPI_Cart_create()` subroutine is tagged with the `CPPC/Nonportable` role. It inserts the appropriate `CPPC_Register_parameter()` calls and adds necessary flow control structures.

4. The compiler performs a safe point detection and inserts a checkpoint in the loop in the `solve()` subroutine. It also performs a live variable analysis and introduces the relevant registers prior to its execution. These include the `i`, `ml`, `a`, `xm`, `ax` and `b` variables. `nl` is not registered, since it is declared as a constant initialized in the code. `r` is an output parameter for the `mpi_dcopy()` call, and hence it is not live at the analysis point. The insertion of a checkpoint inside `solve()` causes calls to this subroutine to be re-executed on restart. Context push and pop function calls are placed enclosing such calls to notify the CPPC library of context changes.

If the application is restarted after a failure, the program flows through relevant blocks recovering the entire application state. Variables are initialized and both the MPI and CPPC initialization functions are called. Then, a conditional jump to label `CPPC_EXEC_0` takes place, transferring execution to the conditional structure that performs the opening of the file `'control.dat'`. Note that the condition is tested before proceeding to REC execution, and so the semantics of the code are preserved. The `CPPC_Register_descriptor()` function ensures that the `iinput` descriptor state is correctly recovered. Next, the communication topology is recreated upon reaching label `CPPC_EXEC_2`. Then the execution moves to the subroutine call on label `CPPC_EXEC_3`. Once inside, relevant variable values are recovered upon reaching label `CPPC_EXEC_4`, and later the checkpoint on label `CPPC_EXEC_5` is reached. CPPC determines that all the state has been restored, and the execution is resumed normally in checkpoint operation mode.

3.4. Implementation details

This section covers the compiler implementation in some detail. Each subsection is focused on a different implementation issue, covering the addition of the necessary features for Cetus to support Fortran 77 codes; the design patterns used to share as much code as possible between the C and Fortran 77 versions of the compiler; and the elegant implementation of code analyses.

3.4.1. Fortran 77 support

The original Cetus provides a compiler research environment that facilitates the development of interprocedural analyses for C applications. There are three basic

layers on the Cetus design: a C parser, that reads source code and creates its associated AST; classes representing the AST itself, providing representations of the code and operations to manipulate it; and a code generator back-end that performs the reverse transformation, reading an AST and outputting a C code.

Our goal when implementing the CPPC compiler was to be able to reuse the compilation passes' code for processing both C and Fortran 77 applications. In order to do so, a front-end capable of building a Cetus-compatible AST representation of a Fortran 77 code was required. This front-end was implemented by writing a Fortran 77 grammar using the ANTLR [46] parser generator tool. It was also necessary to add new classes to the Cetus AST to represent Fortran 77 concepts that do not exist in C. The following abstractions were included on the `cppc.compiler.ast.fortran77` package:

- `COMMON` declarations: A declaration with a block name, and an associated list of regular declarations included in the common block.
- `DATA` declarations: A declaration containing a list of variables and their associated initializers.
- `DIMENSION` declarations: A declaration containing a list of variables and their associated dimensions.
- `IMPLICIT` declarations: A class containing a beginning character, an end character, and an associated specifier to the range.
- `COMPLEX` datatype: A specifier for representing complex datatypes.
- Array specifiers: Fortran array specifiers differ from C ones in that their lower bound may be specified when the array is declared. Since the Cetus version of the array specifier only contains an upper bound, this class was created to allow for the inclusion of lower bound declarations.
- String specifiers: Fortran allows for strings to be defined as `CHARACTER*` variables. Since the concept of string does not exist in C, this class was added.
- Double precision literals: Cetus does only support floating point literals, without specifying their precision. Therefore, all literals should be converted either to floats or doubles when building the AST, which would lead to potential precision issues. This class is used for representing double literals, while the original Cetus one is reserved for floats.

- Substring expressions: Supports substring expressions such as `STRING(A:B)`.
- Implied `DO`: Supports implied `do` constructions such as `print *,(i+1,i=0,N)`.
- Intrinsic I/O calls: Some Fortran intrinsics, such as `print` or `write`, accept a number of regular parameters and then any number of variable arguments, such as in `write(FILE), i, j, k, ...` This class represents such a call, separating its parameters in regular ones (to be included between brackets) and variable ones.
- `FORMAT` statements.
- Computed `GOTO`s: Support for Fortran-style computed `GOTO`s, of the form `GOTO (LABEL1,LABEL2,...) INDEX`.
- `DO` loops: Support for Fortran-style `DO` loops, that have no direct equivalent in C.

Ideally, once the source code is represented as an AST, the compilation passes are able to handle applications regardless of the source language they were originally written in. However, there are certain code structures that have to be handled in different ways for different programming languages. The next subsection covers how these differences are handled.

Once all passes have been applied, a Fortran 77 back-end, created specifically for CPPC, reads the modified AST and generates the Fortran 77 output.

3.4.2. Sharing code between the C and Fortran 77 compilers

Since both the C and the Fortran 77 compiler use the same basic AST, most of the code used for analyses and transformations is usable by the compilers of both programming languages. Some constructs, however, need a slightly different treatment. In order to share as much code as possible, the implementation of the compilation passes uses the template method design pattern [28]. Analyses that need different versions have a common implementation depending on some abstract methods, which are then implented in language-dependent classes. The operations that differ and therefore need these pattern to be used are:

- Insertion of control flow code: Two actions differ when inserting control flow code. First, the number and types of variables inserted to orchestrate the

restart jumps. While the C version uses an array of `void *` values, Fortran 77-style computed gotos do not require such variable. Also, since the Fortran 77 version of `CPPC_JUMP_NEXT()` is a subroutine rather than a function, it adds a `CPPC_JUMP_TEST` variable to store its results. Besides, the conditional jumps code is different due to the differences in computed gotos syntax.

- Language-dependent transforms: As detailed in Section 3.2.8.
- Pointers: All the transformations that depend on whether the accessed variables are pointers or statically allocated are only present in the C version of the compiler. These kind of operations include determining whether a variable is static or dynamic, obtaining the size of a variable, and obtaining the base memory address for a variable.
- CPPC calls: Some CPPC library functions have different signatures for C and Fortran 77. Thus, the code that inserts these calls into the application is different for both compilers.
- Analysis of language-specific AST objects: Some of the representations of the code are specific to one of the languages. Examples of these are implied `do` loops in Fortran 77, or `for` loops in C. The classes in charge of AST analysis, further described in the next subsection, are specialized so that superclasses contain analysis methods for common structures, while the subclasses contain analysis code for language-specific constructs only.

3.4.3. AST analyzers

Performing interprocedural analyses on the entire application is a complex process, that includes many interactions between the different levels of the abstract representation. The AST is composed primarily of objects belonging to two class hierarchies: statements and expressions. Classes in charge of performing interprocedural analyses, such as those for performing the data flow analysis, traverse the tree in a depth-first fashion, beginning with statements and eventually accessing their contained expressions. Thus, they need to have a method for processing each different class in each of the hierarchies. The normal approach for solving this design problem would be to use a visitor pattern [28]. However, this would require both the statements and expressions hierarchies to be modified in order to accept their corresponding visitors. To avoid modifying the Cetus core, a variation

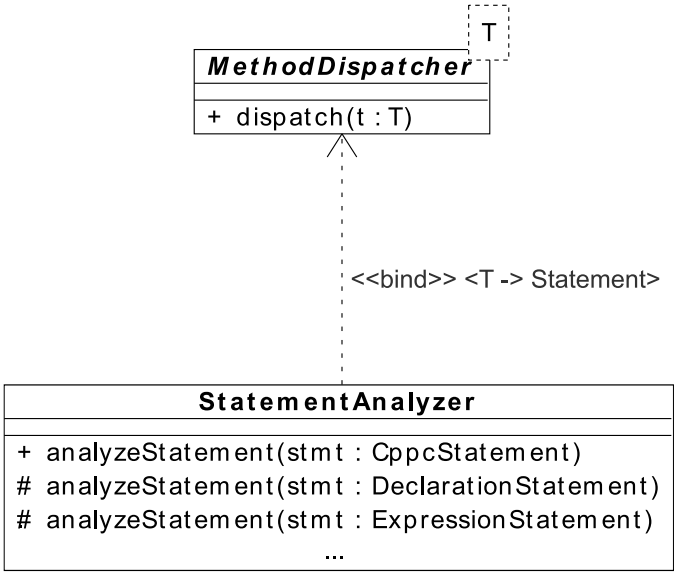


Figure 3.11: AST analyzer implementation by instantiation of the method dispatcher

of the visitor pattern taking advantage of the Java reflection interface was implemented. An example for a class analyzing statements is depicted in Figure 3.11. The `MethodDispatcher` is a an abstract Java Generic implementing a single operation: accessing the Java reflection interface to fetch a method able to process an object of a given class `t` in the `T` hierarchy. AST analyzers are implemented through instantiation of this Generic. For instance, the `StatementAnalyzer` class binds the template parameter `T` to the Cetus `Statement` superclass, and implements methods for analyzing each of its subclasses. The only provided public method accepts a `CppcStatement`, introduced in Section 3.2.4. It fetches the contained statement, and issues a `dispatch()` petition asking for the appropriate method for analyzing it. The dispatcher answers with the only protected method implemented by the `StatementAnalyzer` that accepts an object of the specified class of statement (e.g. `DeclarationStatement`, `ExpressionStatement`, etc.). Thus, visitors for each of the AST hierarchies are implemented without modifications to the original codes.

However, accessing the Java reflection interface is a very inefficient operation. The compiler, in a typical analysis of a parallel code, may perform millions of calls to the dispatch function. In order to alleviate the reflection overhead, the `MethodDispatcher` class includes an associative cache linking subclasses of `T` and their corresponding processing methods. Since the target hierarchies do not have a large number of members the cache will quickly be filled, effectively neglecting the

penalization for using the reflection interface.

3.5. Related work

Most checkpointing approaches in the literature perform structural analyses and source code modifications to instrument checkpointing insertion. Many of them, however, leave the insertion of checkpoints to be manually performed by the user, while automatically performing the remaining instrumentation (variable storage, recovery, control flow, etc.). Porch [51] and C^3 [13–15] require that the user inserts checkpoint calls in the code. These calls will only trigger an actual checkpoint according to a frequency timer. These "potential checkpoints" were originally introduced by CATCH GCC [39], which also automated their insertion. A potential checkpoint was introduced at the beginning of subroutines, and at the first line inside a loop. This checkpoint placement guaranteed, in the general case, that potential checkpoint calls would be executed often enough so as to provide a checkpointing frequency reasonably close to the desired one. However, this approach cannot be followed by CPPC, where checkpointing frequencies are not defined in temporal terms, due to the need to statically coordinate all processes independently of how long they take to progress through the application's execution.

For checkpointing schemes that do not use runtime coordination, such as CPPC, checkpoints have to be inserted at places where the global consistency of the approach is guaranteed. When working with implicitly parallel languages, the compiler is able to use the native constructions for parallelism to extract information about safe places for checkpointing. Such is the case of the work in ZPL by Choi et al. [20], where safe communication-free checkpoint ranges are detected, and a checkpoint inserted at the single location in the range with the fewest live variables.

When working with explicit communication interfaces, such as MPI, the communication pattern is user-defined. The compiler has to interpret the communication statements inserted in the code, and perform a communication matching to find safe locations for checkpointing. Some analysis models have been developed with the goal of testing the correctness of the communication pattern in an application. However, pursuing correctness verification is a tougher problem than finding safe regions in the code, and these models typically present strong limitations, such as not being able to deal with non-blocking communications or wildcard receives (`MPI_ANY_SOURCE` and `MPI_ANY_TAG`) [59].

Performing static communication analyses typically involves the use of MPI control flow graphs [58], that extend the concept of a control flow graph with the introduction of communication edges that connect communication nodes. Since correctness tests are not one of their goals, they assume that the communications are correct and conservatively represent indeterminacies by drawing all possible communication edges between the nodes involved. A node in the graph is considered to be a safe point if there is no communication edge connecting one of its ancestors with one of its successors. CPPC uses the same approach as described for the analysis, except that no graph is used. Since our goal is not to obtain a representation of the communication pattern in the application, but rather, to simply categorize points in the code as either safe or non-safe, the graph may be ommitted, and the communication buffer object previously described used for representing pending communications at each point in the code. Not building the graph has several advantages in terms of efficiency. Mainly, it minimizes memory consumption and does not require the analysis of the graph resulting from the compilation pass.

With regards to the automatic checkpoint insertion, some theoretical approaches calculating the optimal mathematical solution to the checkpoint placement problem exist [67, 69]. However, these assume that all involved parameters are known. This is not the case under CPPC, where the execution environment is not assumed to be fixed due to its inherent characteristic of portability.

Checkpoint file sizes are tightly related to scalability issues when checkpointing parallel applications. Thus, different approaches to reduce the amount of data stored in state files have been studied during the last decade. Plank et al. proposed in [48] a memory exclusion technique based on performing a dead variable analysis to identify memory regions which can be safely excluded from the checkpoint process. CPPC implements a similar technique, based on performing a live variable analysis at compile time to identify those variables that should be included in the checkpoint file.

3.6. Summary

This chapter has covered both the design and implementation of the CPPC source-to-source compiler, in charge of automating the integration of a parallel or sequential application with the CPPC library. It is implented in Java, offering widespread portability, using the Cetus framework, which has been extended to

support Fortran 77 in addition to C codes. The compiler performs all necessary analyses and transformations to the code, including static data flow, communication analyses and the insertion of checkpoints at key places in the code. All these analyses have been thoroughly described, as well as their implementation details focusing on the design decisions taken to enable the reusability of the platform. By using a semantic catalog, the compiler is able to analyze a variety of APIs for communications, file I/O, etc., and thus it is not tied to any specific interface such as MPI. Concluding the chapter, the related work has been covered, giving an overview of other existing techniques for checkpoint insertion and communication analysis.

Chapter 4

Experimental Results

4.1. Introduction

In order to conduct the experimental evaluation of the CPPC checkpointing framework, two different sets of experiments were designed and executed to separately assess the performance of both the CPPC compiler and the CPPC library. Twelve applications were selected for testing, separated into three different types. First, the eight applications in the NPB-MPI v3.1 benchmarks [3]. These have short runtimes, which makes them ill-suited choices for checkpoint insertion. However, they are well-known and widespread, which makes them a good choice for comparison purposes. The next two programs are scientific applications in use in the Galician Supercomputing Center (CESGA), called *CalcuNetw* and *Fekete*. While these applications do not have particularly long runs, they are good choices for evaluating the performance of the framework in applications currently in use in supercomputing centers. Finally, two large-scale applications called DBEM and STEM were also added to test the tool in long running programs. Table 4.1 gives more details about the test applications used.

This chapter is divided into two sections. Section 4.2 covers experimental results for the CPPC compiler. Section 4.3 details execution tests for the CPPC library.

Table 4.1: Summary of test applications

Source	Application	Description
NAS NPB-MPI v3.1	BT	A simulated CDF application that uses an implicit algorithm to solve 3-dimensional compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternate Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions. The resulting systems are **B**lock-**T**ridiagonal of $5x5$ blocks and are solved sequentially along each dimension.
	CG	Uses a **C**onjugate **G**radient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix.
	EP	**E**mbarrassingly **P**arallel benchmark. It generates pairs of Gaussian random deviates according to a specific scheme.
	FT	Contains the computational kernel of a fast **F**ourier **T**ransform (FFT)-based spectral method. It performs three 1-dimensional FFT's, one for each dimension.
	IS	**I**nteger **S**ort: It works with a list of small integer values, not really sorting them but assigning every list member a number indicating the position in the sorted list.
	LU	A simulated CFD application that uses a symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the 3-dimensional Navier-Stokes equations by splitting it into block **L**ower and **U**pper triangular systems.
	MG	Uses a V-cycle **M**ulti**G**rid to compute the solution of the 3-dimensional scalar Poisson equation. The algorithm works continuously on a set of grids that are made between coarse and fine.

Table 4.1: Summary of test applications (continued)

Source	Application	Description
NAS NPB-MPI v3.1	SP	A simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system has **S**calar **P**entadiagonal bands of linear equations that are solved sequentially along each dimension.
CESGA	CalcuNetw [43]	Calculates some characterization measurements in a given network, consisting of a set of nodes or vertices joined together in pairs by links or edges, and compares it with a number of random networks specified by the user. The program calculates the subgraph centrality (SC), SC odd, SC even, bipartivity, network communicability, and network communicability for connected nodes.
	Fekete [17]	Determines the position of a certain number of points on a 2-dimensional sphere such that the potential energy produced by the interaction of these points is minimum. This is the 7th of the Smale's problems [60].
	DBEM [30]	Crack growth analysis using the **D**ual **B**oundary **E**lement **M**ethod. The analysis leads to a large number of discretized equations that grow at every step when the crack growth is evaluated. It solves the resulting dense linear system using the GMRES iterative method, regarded as the most robust of the Krylov subspace iterative methods.
	STEM-II [42]	Used to know in advance how the meteorological conditions, obtained from a meteorogical prediction model, would affect the emissions of pollutants by the power plant of As Pontes (A Coruña, Spain) in order to fulfill EU regulations. The underlying equation is a time-dependent, 3-dimensional partial differential atmosferic-diffusion equation.

4.2. Compiler

This section focuses on two main issues when using the CPPC compiler to instrument parallel applications: correction of the resulting code, covered in Section 4.2.1; and the performance obtained when executing the necessary analyses and transformations, detailed in Section 4.2.2.

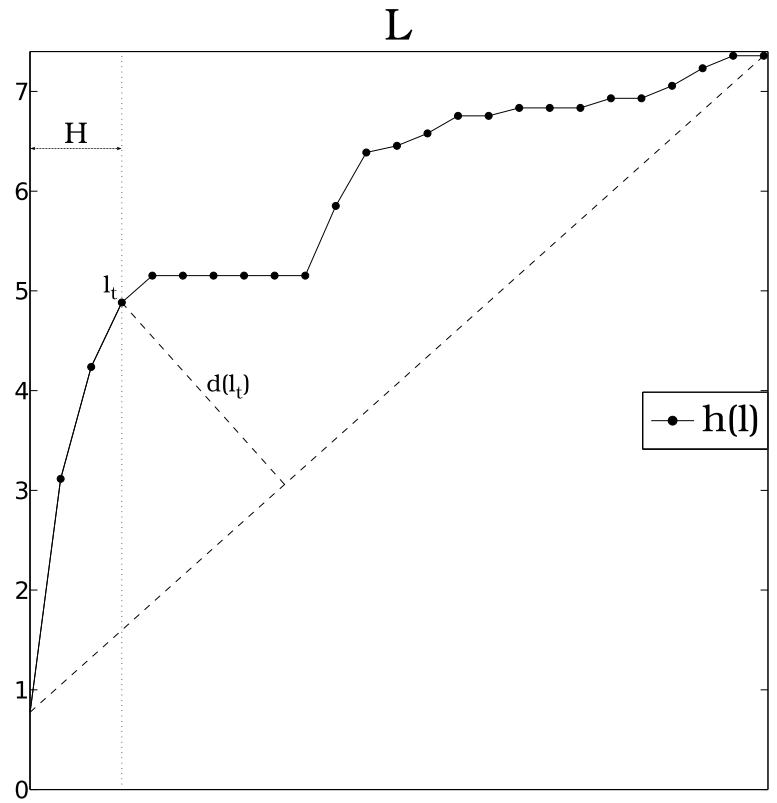
4.2.1. Analysis of the instrumented codes

In this section, the results of the checkpoint insertion analyses are shown. The following subsections detail the results of the first and second thresholding steps of the checkpoint insertion algorithm for all the test applications, and compare them to the optimal manual checkpoint placement performed by the authors during the initial tests of the applications. While some extra checkpoints are inserted in applications with similar heuristic values for both time-consuming loops and negligible ones, the compiler never leaves a time-consuming loop without checkpointing. This means that the analysis behaves in a conservative way. Thus, while the resulting code may not be completely optimal, application progress is guaranteed.

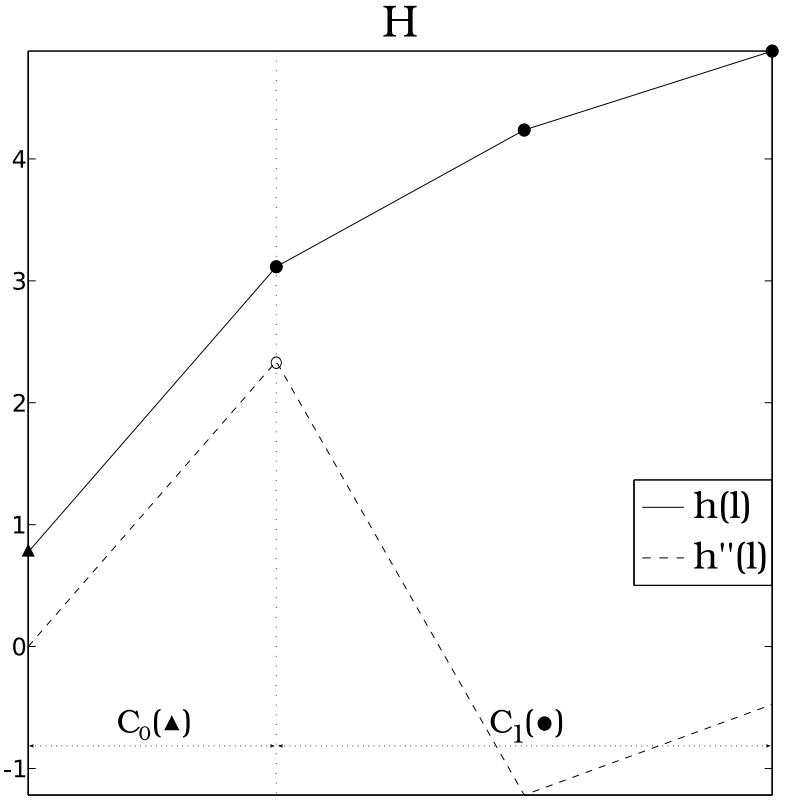
Regarding the communication matching, the compiler correctly detects safe points for all test applications.

- NAS BT

Figure 4.1(a) shows the heuristic $h(l)$ as calculated for the NAS BT application. The figure is the same as the one shown for exemplifying the checkpoint insertion algorithm described in Section 3.2.7. The first step of the thresholding process selects a subset of four loop nests as candidates for checkpoint insertion. The values of $h(l)$ for these loops are detailed in Table 4.2. Loop #1 corresponds to the main computational loop in BT, and is the place where a manual checkpoint was inserted by the authors in the preliminary application tests. The second thresholding step, shown in Figure 4.1(b), determines that the loops in C_0 are responsible for the 56.94% of the total variation of $h(l)$ in the H subset, and therefore selects only loop #1 for checkpoint insertion, which matches the optimal checkpoint insertion in the manual analysis.



(a) First step: shape-based threshold



(b) Second step: cluster-based threshold

Figure 4.1: Checkpoint insertion for NAS BT

Table 4.2: Detail of loops selected by the shape-based threshold for NAS BT

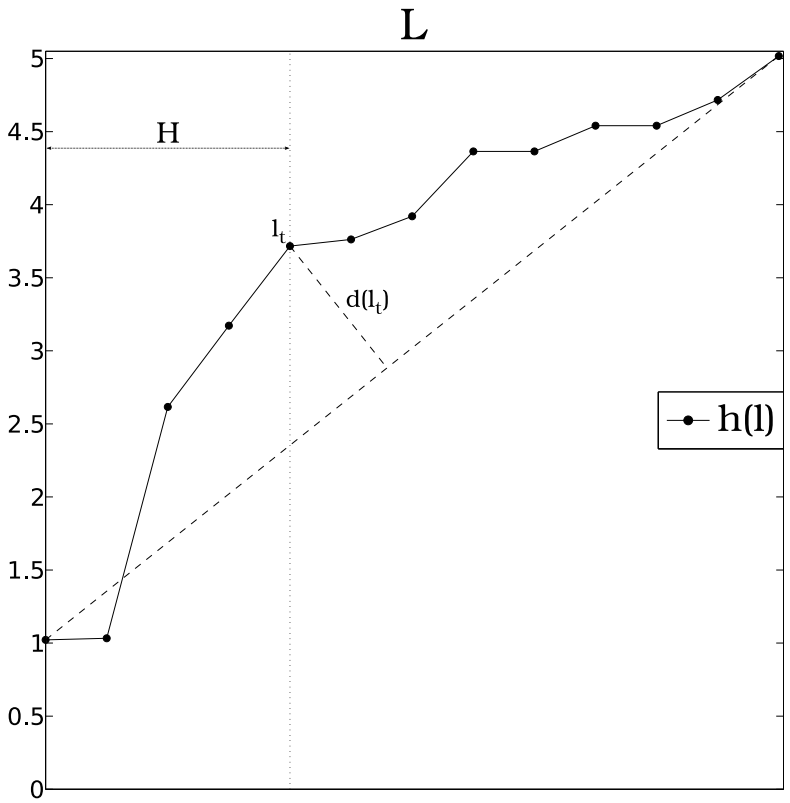
Loop #	**File**	**Line**	**Statements**	**Accesses**	$h(l)$
1	`bt.f`	179	2075	5547	0.7755
2	`exact_rhs.f`	24	107	490	3.1149
3	`initialize.f`	44	30	132	4.2368
4	`error.f`	25	19	47	4.8837
Total program statements					5084
Total program accesses					13502

- NAS CG

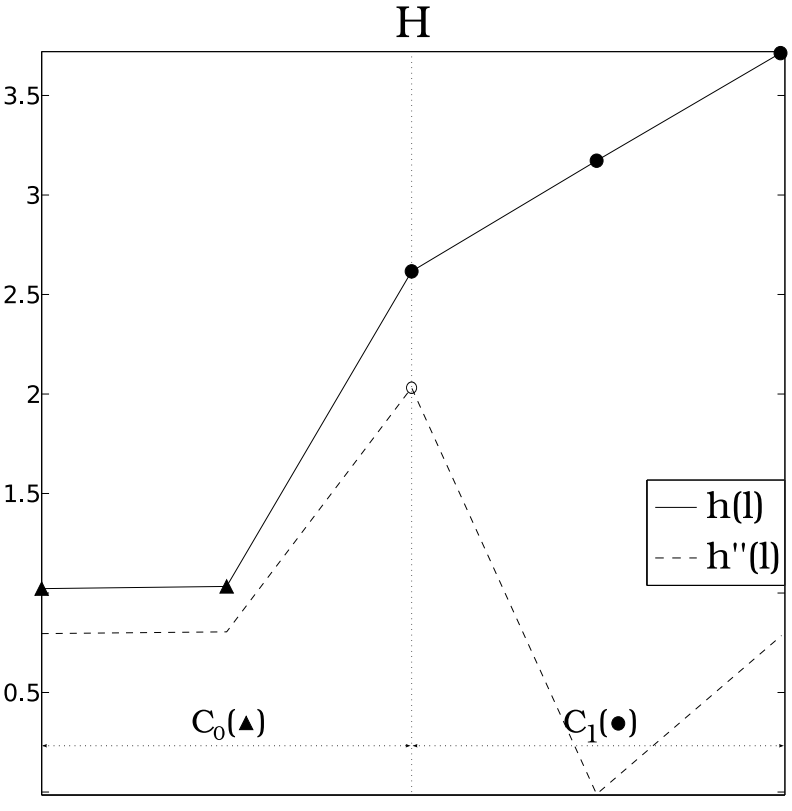
Figure 4.2(a) shows the heuristic $h(l)$ calculated for the NAS CG application. The first step of the thresholding process selects a subset of five loop nests as candidates for checkpoint insertion. The values of $h(l)$ for these loops are detailed in Table 4.3. Loop #1 corresponds to the main computational loop in CG, and is the place where a manual checkpoint was inserted by the authors in the preliminary application tests. The second thresholding step, shown in Figure 4.2(b), creates two different clusters, and selects C_0 for checkpointing, responsible for 59.16% of the total variation of $h(l)$ in the H subset. Two loops are included in C_0, loop #2 being a exact copy of loop #1. The only difference is that it performs a single untimed iteration which initializes all code and data page tables, and that it does not perform the conditional statement found in line 500 in file `cg.f`. This accounts for the difference in $h(l)$ for both loops, which is negligible as shown in the tables. After executing this single-iteration loop, the application reinits and starts timing. The compiler was not programmed to detect single-iteration loops as not real loops, and therefore this loop is selected as well for checkpointing, which may create an additional redundant checkpoint depending on the runtime configuration of the library, particularly on whether or not the user decides to activate checkpointing each time a new checkpoint is reached, regardless of the number of calls performed to the checkpoint function.

- NAS EP

Figure 4.3(a) shows the heuristic $h(l)$ as calculated for the NAS EP application. The first step of the thresholding process selects a subset of two loop nests as candidates for checkpoint insertion. The values of $h(l)$ for these loops are detailed in



(a) First step: shape-based threshold



(b) Second step: cluster-based threshold

Figure 4.2: Checkpoint insertion for NAS CG

Table 4.3: Detail of loops selected by the shape-based threshold for NAS CG

Loop #	**File**	**Line**	**Statements**	**Accesses**	$h(l)$
1	`cg.f`	441	97	220	1.0220
2	`cg.f`	344	96	217	1.0329
3	`cg.f`	1453	28	20	2.6160
4	`cg.f`	1606	14	10	3.1723
5	`cg.f`	910	4	10	3.7163
Total program statements					354
Total program accesses					647

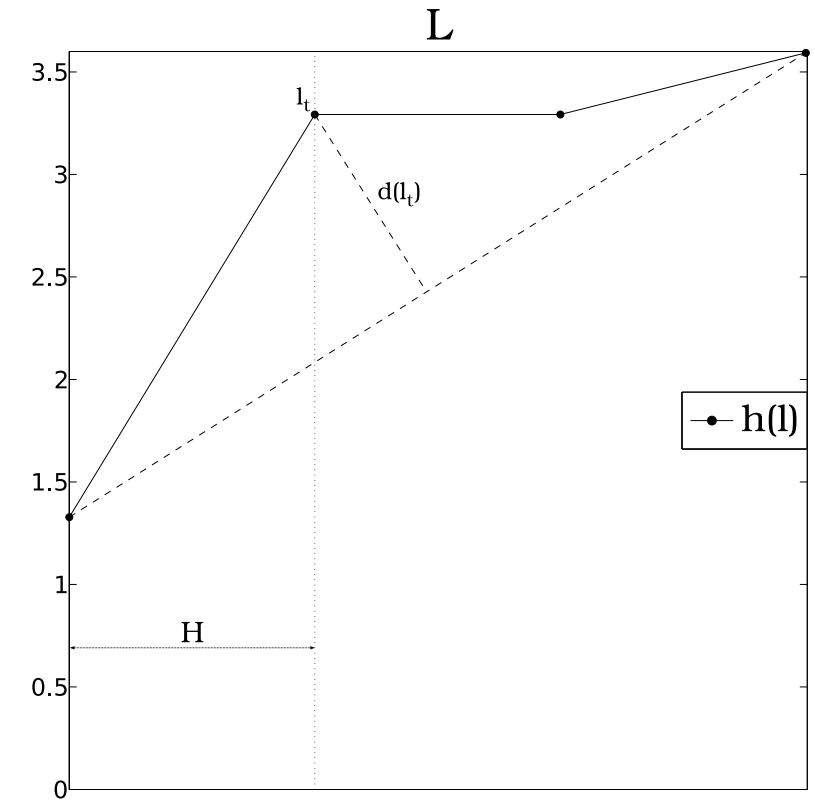
Table 4.4: Detail of loops selected by the shape-based threshold for NAS EP

Loop #	**File**	**Line**	**Statements**	**Accesses**	$h(l)$
1	`ep.f`	189	16	23	1.3285
2	`ep.f`	165	2	2	3.2923
Total program statements					70
Total program accesses					114

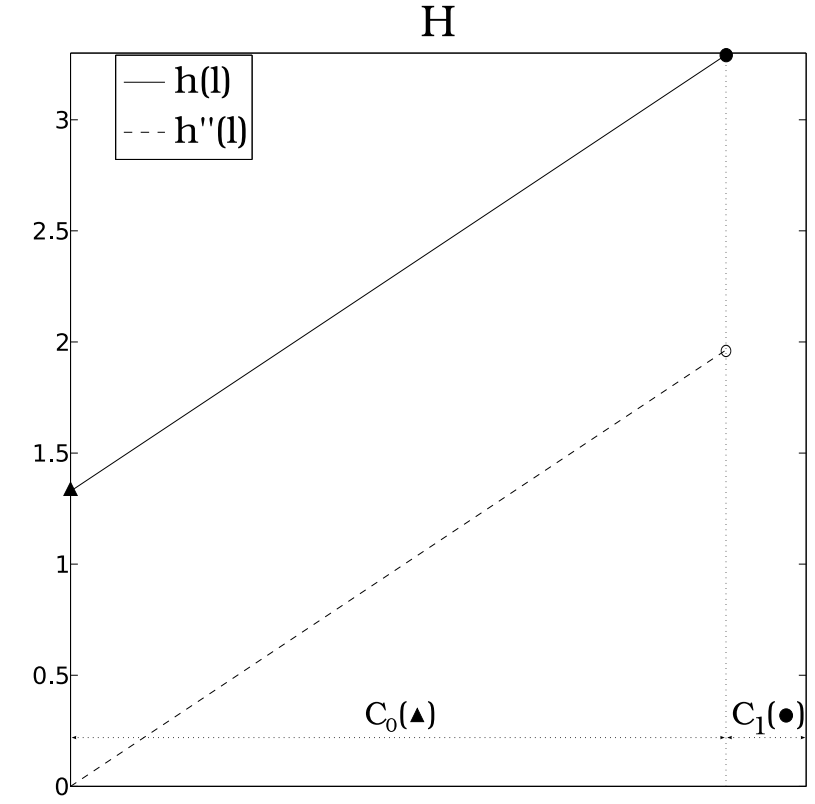
Table 4.4. Loop #1 corresponds to the main computational loop in EP, and is the place where a manual checkpoint was inserted by the authors in the preliminary application tests. The second thresholding step, shown in Figure 4.3(b), assigns each point in H to a different cluster, and selects loop #1 for checkpoint insertion, responsible for 100% of the total variation of $h(l)$ in the H subset. Note that any situation in which only two points reach the cluster-based thresholding step will result in the algorithm selecting only the one with the bigger $h(l)$ value. This makes perfect sense, since the shape-based threshold would not select only two loop nests if they had similar heuristic values.

- NAS FT

Figure 4.4(a) shows the heuristic $h(l)$ as calculated for the NAS FT application, which is larger and structurally more complex than the two previous ones. The first step of the thresholding process selects a subset of six loop nests as candidates for checkpoint insertion. The values of $h(l)$ for these loops are detailed in Table 4.5. Loop #1 corresponds to the main computational loop in FT, and is the place where a manual checkpoint was inserted by the authors in the preliminary application tests. The second thresholding step, shown in Figure 4.4(b), creates three different



(a) First step: shape-based threshold



(b) Second step: cluster-based threshold

Figure 4.3: Checkpoint insertion for NAS EP

Table 4.5: Detail of loops selected by the shape-based threshold for NAS FT

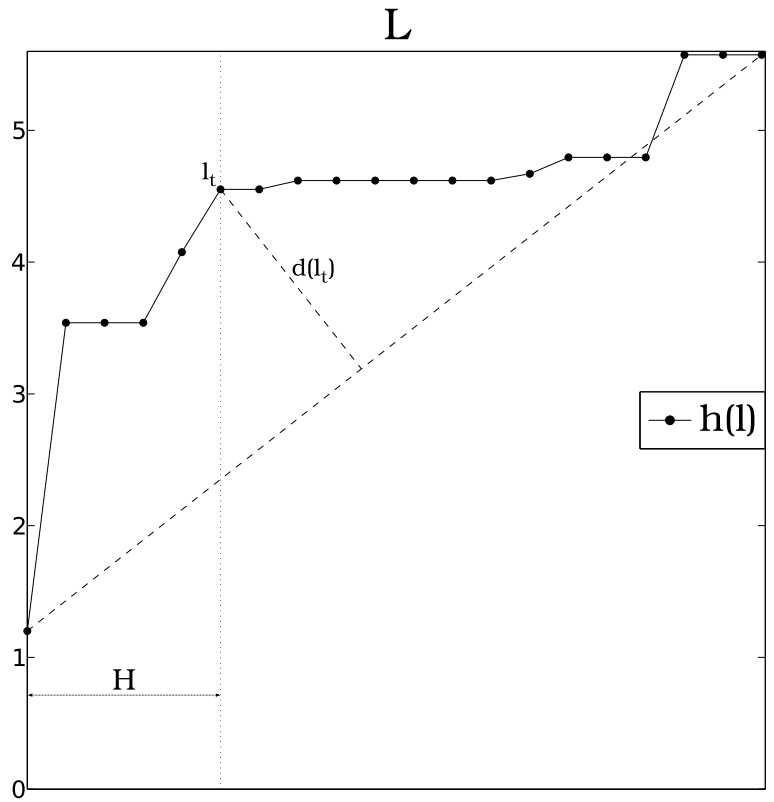
Loop #	**File**	**Line**	**Statements**	**Accesses**	$h(l)$
1	`ft.f`	159	189	337	1.2005
2	`ft.f`	676	9	24	3.5398
3	`ft.f`	689	9	24	3.5398
4	`ft.f`	702	9	24	3.5398
5	`ft.f`	1016	7	9	4.0750
6	`ft.f`	271	3	7	4.5521
Total program statements					743
Total program accesses					1261

clusters, and selects only loop #1 for checkpoint insertion as responsible for 69.80% of the total variation of $h(l)$ in the H subset, which matches the optimal checkpoint insertion in the manual analysis.

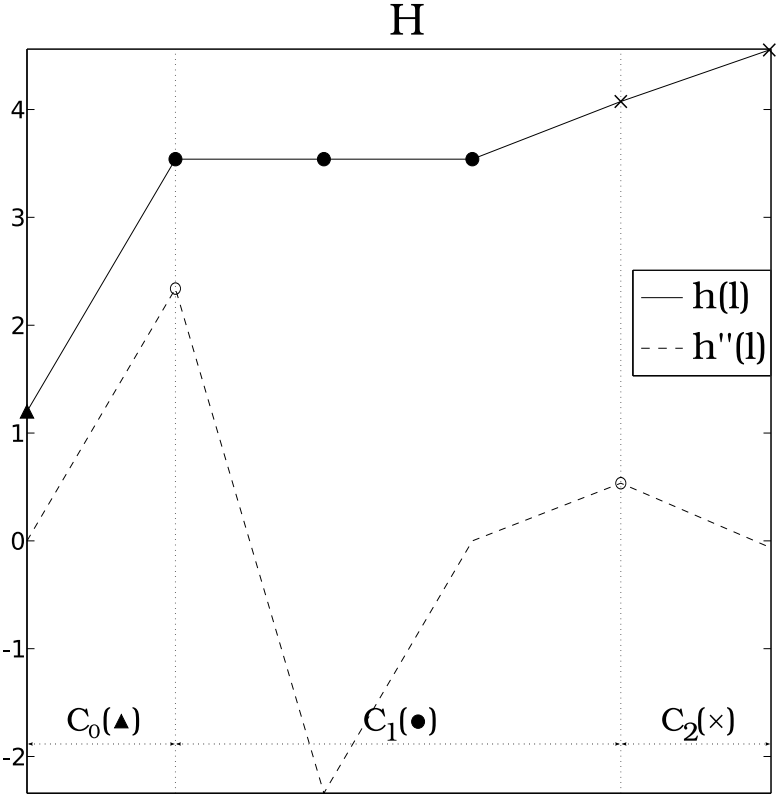
- NAS IS

Figure 4.5(a) shows the heuristic $h(l)$ as calculated for the NAS IS application, the only of the NPB ones written in C. As can be seen, the shape of $h(l)$ is quite atypical, due to the fact that this application is very small and all loops have similar sizes. In fact, this is the application with the smallest total variation in $h(l)$, and the one in which $h(l)$ most closely resembles a straight line. This causes the first step of the thresholding process to select a high number of loop nests as candidates for checkpoint insertion, five out of the total six in IS. The values of $h(l)$ for these loops are detailed in Table 4.6. Loop #2 corresponds to the main computational loop, and the only one manually selected in the preliminary application tests. The second thresholding step, shown in Figure 4.5(b), creates three different clusters. It does not select C_0 only, since it is only responsible for 41.41% of the total variation of $h(l)$ in the H subset. Instead, it selects both C_0 and C_1, which are together responsible for 61.94% of such variation.

Although the checkpoint insertion algorithm does indeed select the main computational loop for checkpoint insertion, thus guaranteeing execution progress in the presence of failures, two other loops are conservatively selected in addition to it, making it a non-optimal choice.

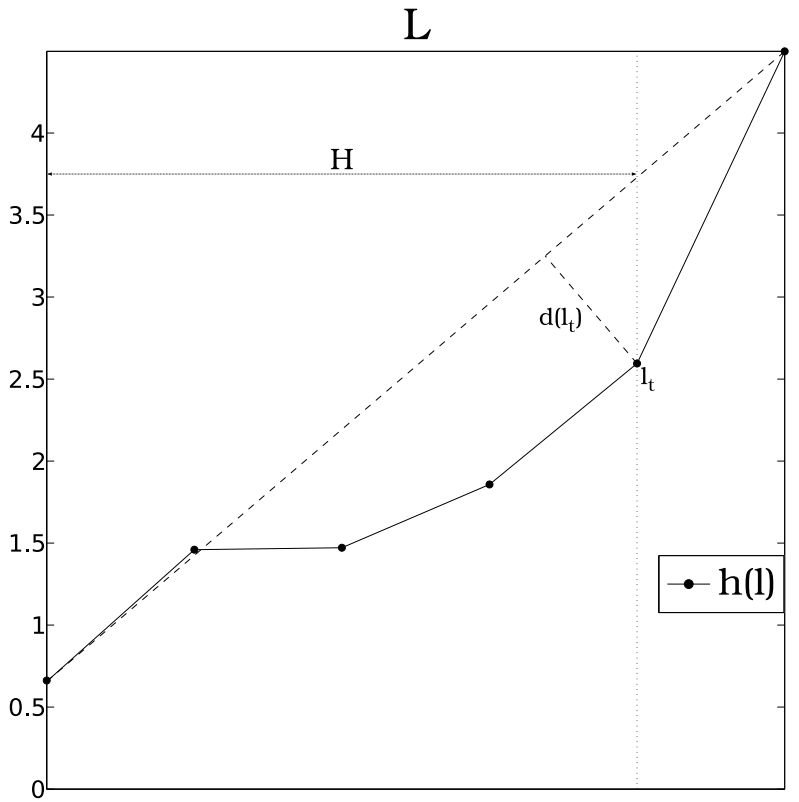


(a) First step: shape-based threshold

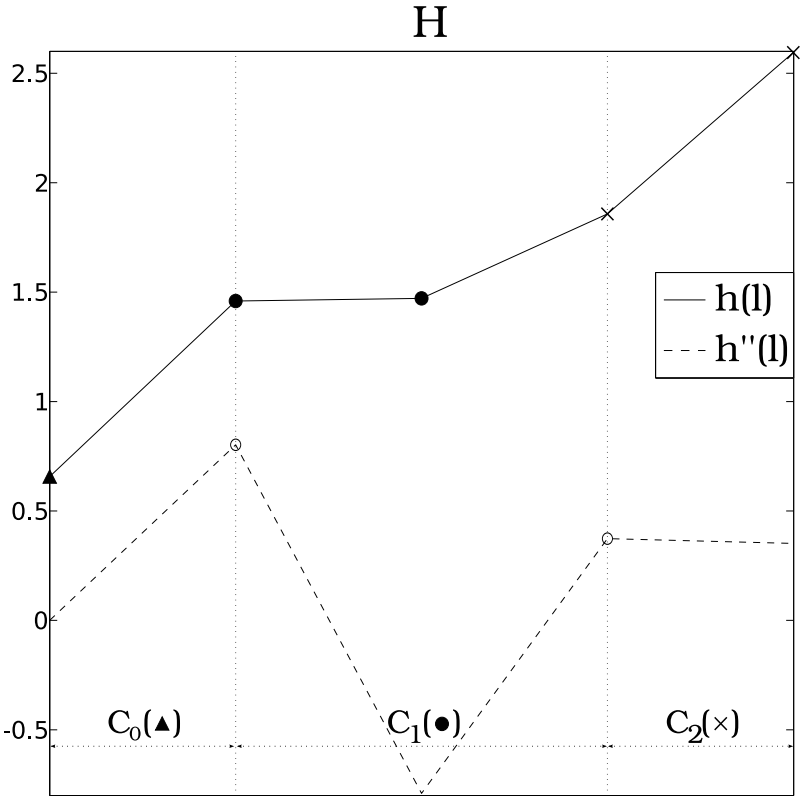


(b) Second step: cluster-based threshold

Figure 4.4: Checkpoint insertion for NAS FT



(a) First step: shape-based threshold



(b) Second step: cluster-based threshold

Figure 4.5: Checkpoint insertion for NAS IS

Table 4.6: Detail of loops selected by the shape-based threshold for NAS IS

Loop #	**File**	**Line**	**Statements**	**Accesses**	$h(l)$
1	`is.c`	425	90	154	0.6570
2	`is.c`	976	56	39	1.4595
3	`is.c`	396	36	59	1.4716
4	`is.c`	387	23	38	1.8573
5	`is.c`	882	16	10	2.5947
Total program statements					242
Total program accesses					260

Table 4.7: Detail of loops selected by the shape-based threshold for NAS LU

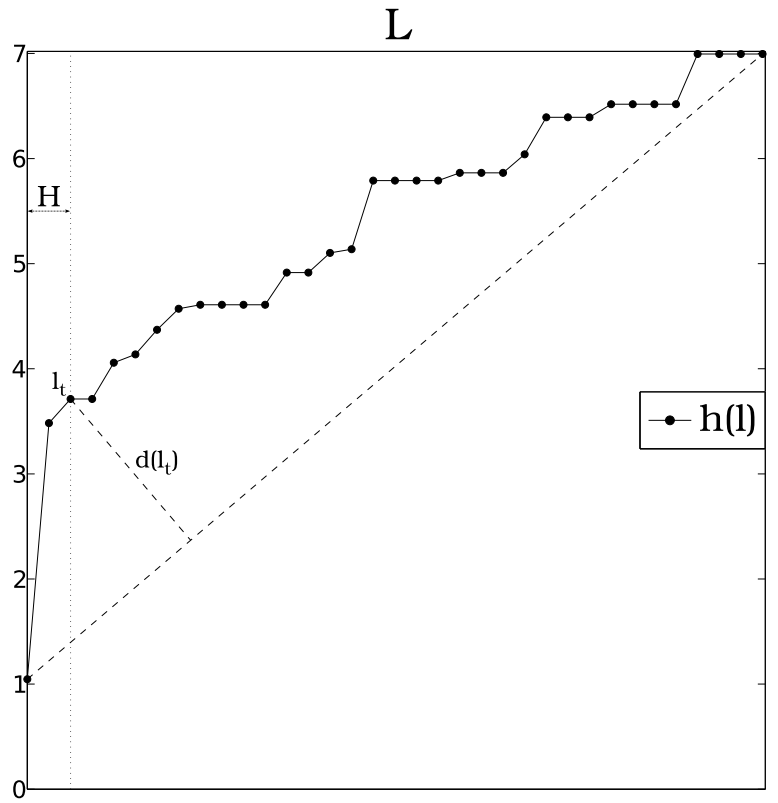
Loop #	**File**	**Line**	**Statements**	**Accesses**	$h(l)$
1	`ssor.f`	78	452	2251	1.0466
2	`erhs.f`	383	43	151	3.4828
3	`erhs.f`	118	33	116	3.7122
Total program statements					1961
Total program accesses					7415

- NAS LU

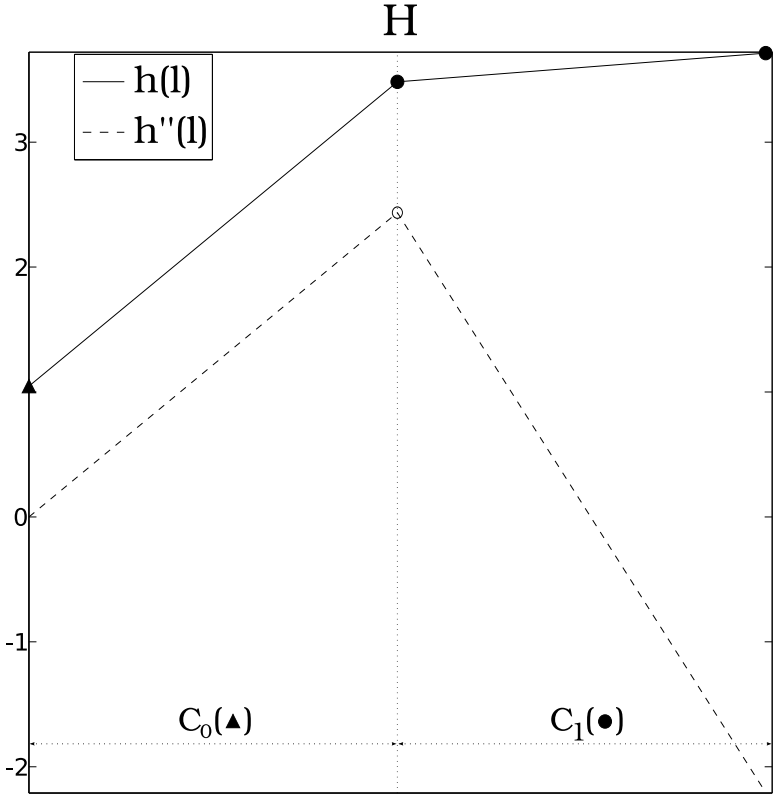
Figure 4.6(a) shows the heuristic $h(l)$ as calculated for the NAS LU application. The first step of the thresholding process selects a subset of three loop nests as candidates for checkpoint insertion. The values of $h(l)$ for these loops are detailed in Table 4.7. Loop #1 corresponds to the main computational loop in LU, and is the place where a manual checkpoint was inserted by the authors in the preliminary application tests. The second thresholding step, shown in Figure 4.6(b), creates two clusters, and selects only loop #1 for checkpoint insertion as responsible for 91.39% of the total variation of $h(l)$ in the H subset, which matches the optimal checkpoint insertion in the manual analysis.

- NAS MG

Figure 4.7(a) shows the heuristic $h(l)$ as calculated for the NAS MG application. The first step of the thresholding process selects a subset of two loop nests as candidates for checkpoint insertion. The values of $h(l)$ for these loops are detailed in Table 4.8. Loop #1 corresponds to the main computational loop in MG, and is the



(a) First step: shape-based threshold



(b) Second step: cluster-based threshold

Figure 4.6: Checkpoint insertion for NAS LU

Table 4.8: Detail of loops selected by the shape-based threshold for NAS MG

Loop #	**File**	**Line**	**Statements**	**Accesses**	$h(l)$
1	`mg.f`	245	503	1354	0.9707
2	`mg.f`	2100	27	46	3.5997
Total program statements					1597
Total program accesses					3862

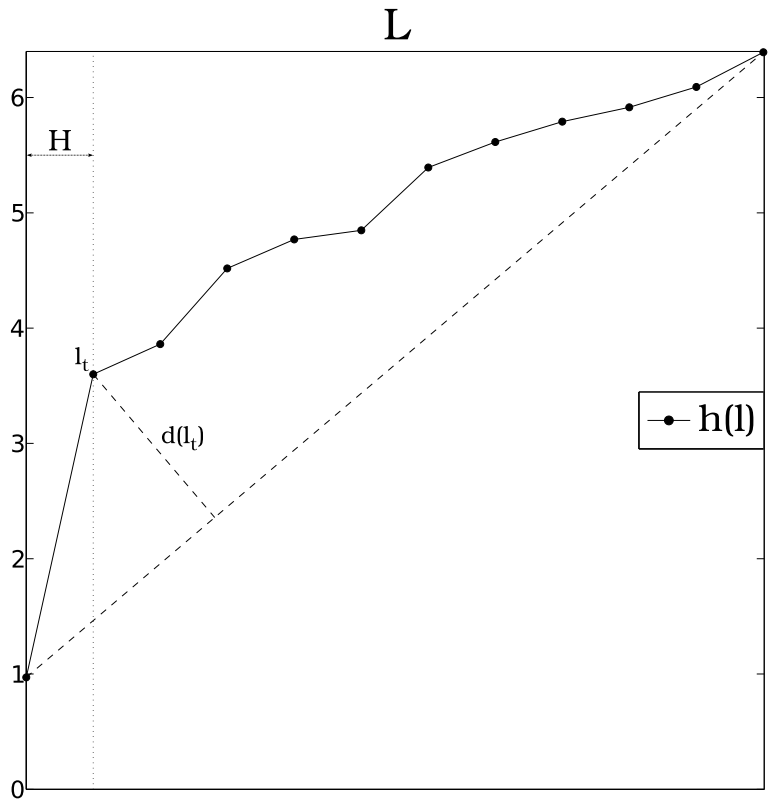
Table 4.9: Detail of loops selected by the shape-based threshold for NAS SP

Loop #	**File**	**Line**	**Statements**	**Accesses**	$h(l)$
1	`sp.f`	150	927	1783	1.0077
2	`exact_rhs.f`	23	107	490	2.5044
3	`initialize.f`	45	30	132	3.6262
4	`error.f`	26	19	47	4.2731
5	`initialize.f`	103	12	40	4.5427
Total program statements					2801
Total program accesses					6009

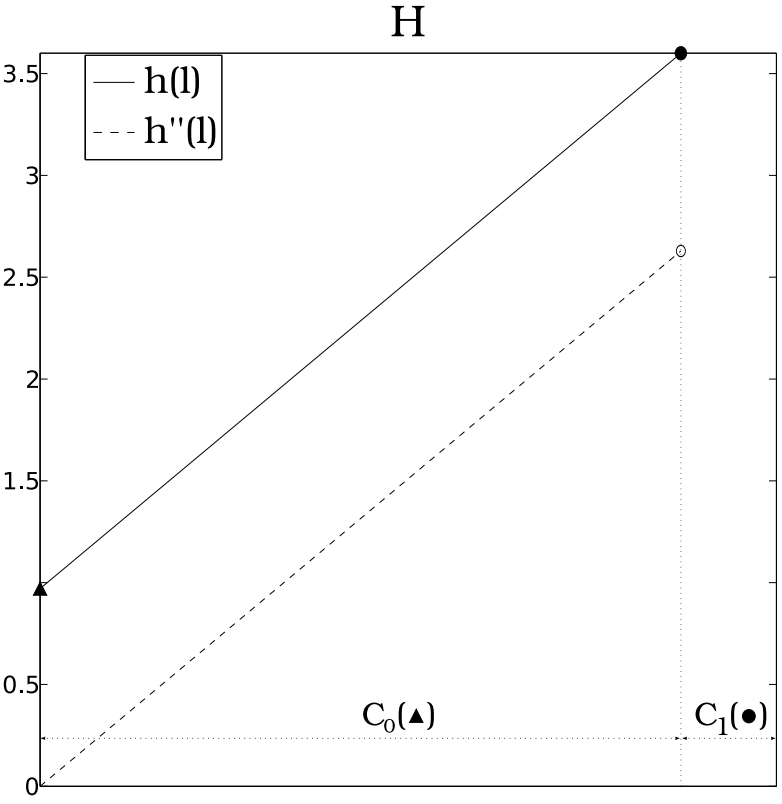
place where a manual checkpoint was inserted by the authors in the preliminary application tests. As in the NAS EP application, the second thresholding step, shown in Figure 4.7(b) creates two clusters, and selects only one loop for checkpointing which is responsible for 100% of the variation of $h(l)$ in the H subset. This behavior matches the optimal checkpoint insertion in the manual analysis.

- NAS SP

The structure of the NAS SP application is very similar to that of the NAS BT, thus the outcome of the checkpoint insertion analysis is very similar to the outcome for the former. The first and second thresholding steps are shown in Figures 4.8(a) and 4.8(b), respectively. Values for $h(l)$ in the H subset, which contains five loops, are detailed in Table 4.9. The cluster-based thresholding step creates three clusters. C_0 is only responsible for a 42.34% of the total variation of $h(l)$ in the H subset, so both C_0 and C_1 get checkpointed, accounting for a 92.37% of the variation of $h(l)$. Manual checkpointing insertion is limited to the loop nest in C_0, which means that all three loops in C_1 are conservatively checkpointed.

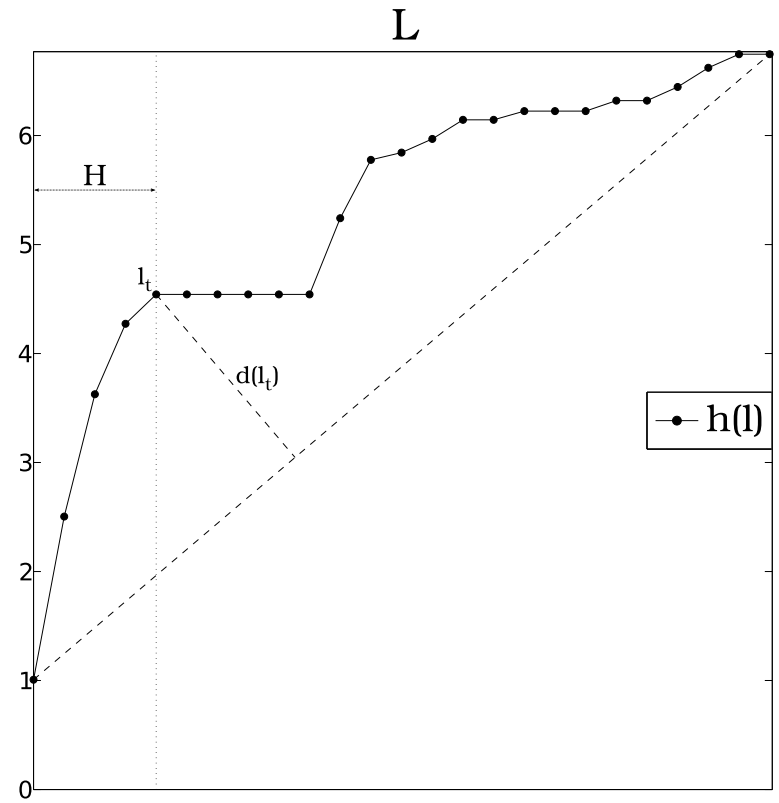


(a) First step: shape-based threshold

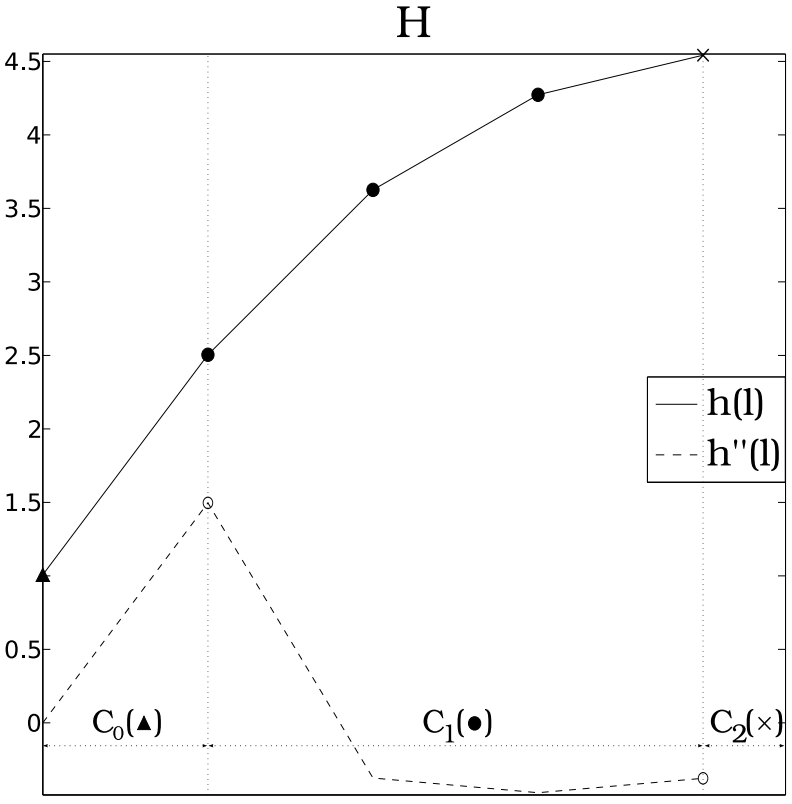


(b) Second step: cluster-based threshold

Figure 4.7: Checkpoint insertion for NAS MG



(a) First step: shape-based threshold



(b) Second step: cluster-based threshold

Figure 4.8: Checkpoint insertion for NAS SP

Table 4.10: Detail of loops selected by the shape-based threshold for CESGA CalcuNetw

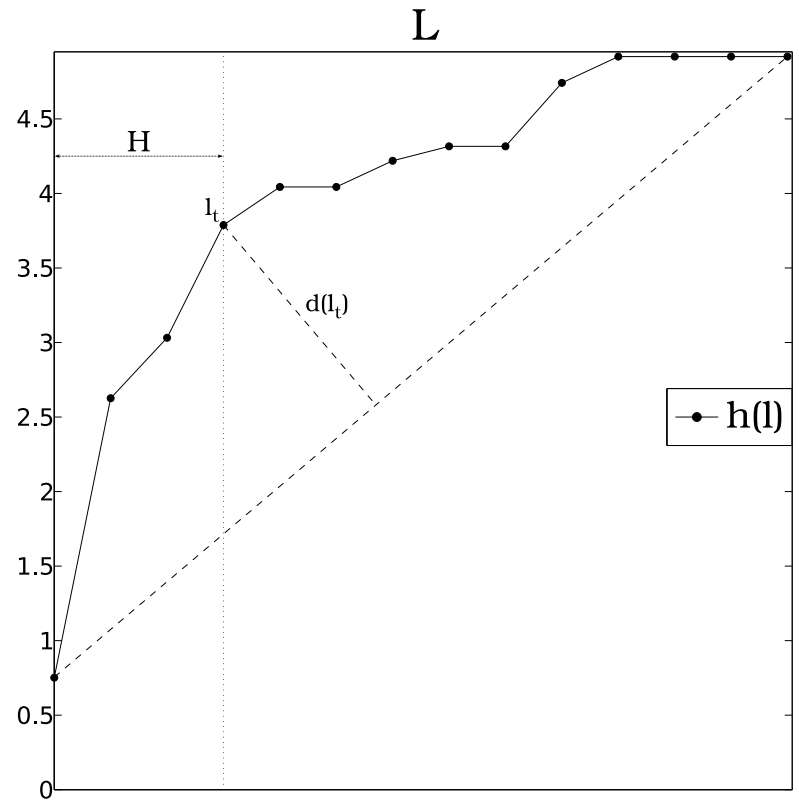
Loop #	**Statements**	**Accesses**	$h(l)$
1	162	180	0.7548
2	28	14	2.6263
3	14	11	3.0321
4	9	3	3.7883
Total program statements			392
Total program accesses			423

- CESGA CalcuNetw

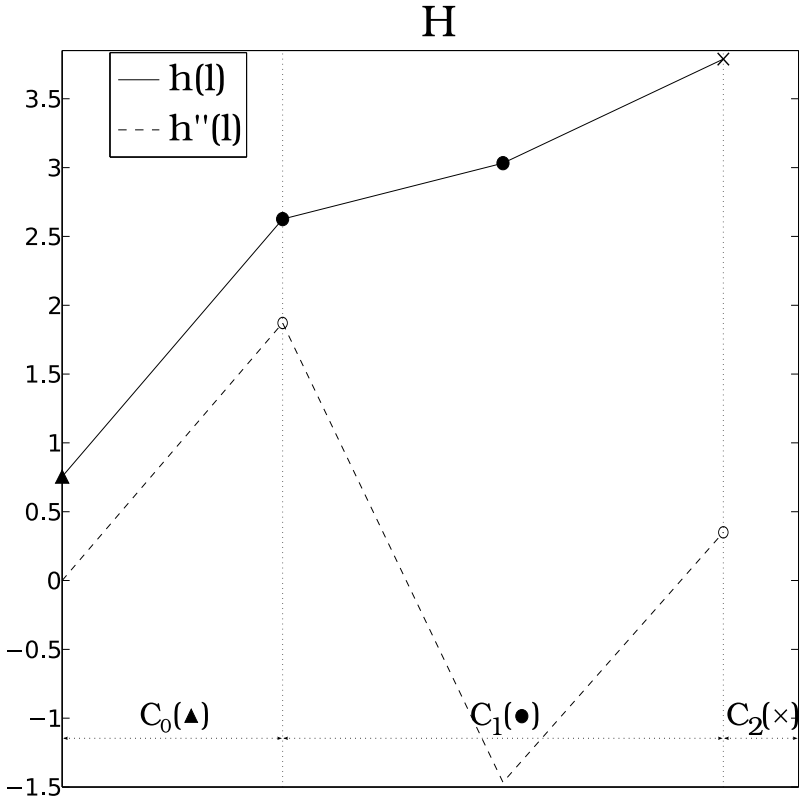
The code of the following applications is not in the public domain, and therefore it makes no sense to give details about code locations of the selected loops. Heuristic values for loop nests in the H subset, however, are included in Table 4.10 for completitude. It is interesting to detail the shape of $h(l)$ and how the insertion algorithm behaves, since these are naturally better candidates for checkpoint insertion than the NAS Parallel Benchmarks. Figure 4.9(a) shows the first step of the thresholding algorithm, while Figure 4.9(b) depicts the second one. As can be seen, the shape of the application is similar to the larger ones in the NPB set. The shape-based threshold selects a subset of four loop nests as candidates for checkpoint insertion, which are categorized in three different clusters by the second thresholding step. Loop #1 is then selected for checkpoint insertion as it contains 61.70% of the variation of $h(l)$ in the H subset. This loop is the main computational loop in the application, and the decision matches the manual one performed during the initial assessment of the application.

- CESGA Fekete

Again, the automatic analysis for this application matches the outcome of the manual decision process. The first and second thresholding steps are shown in Figures 4.10(a) and 4.10(b), respectively. Values for $h(l)$ in the H subset are detailed in Table 4.11. Two clusters are created in the second thresholding step. Loop #1 in C_0 is the only nest to be checkpointed, being responsible for 68.84% of the total variation of $h(l)$ in the H subset.

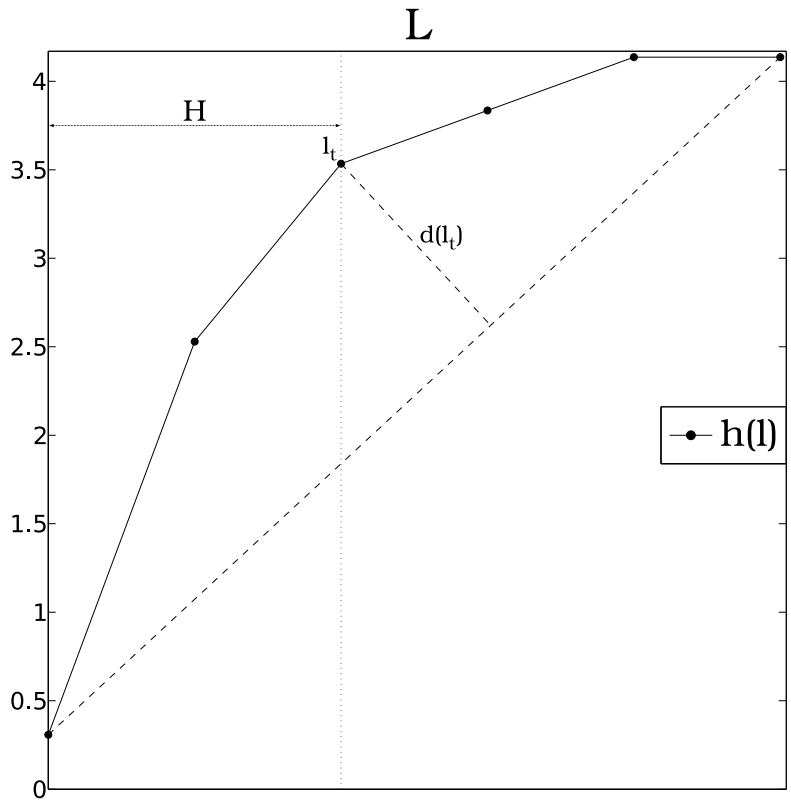


(a) First step: shape-based threshold

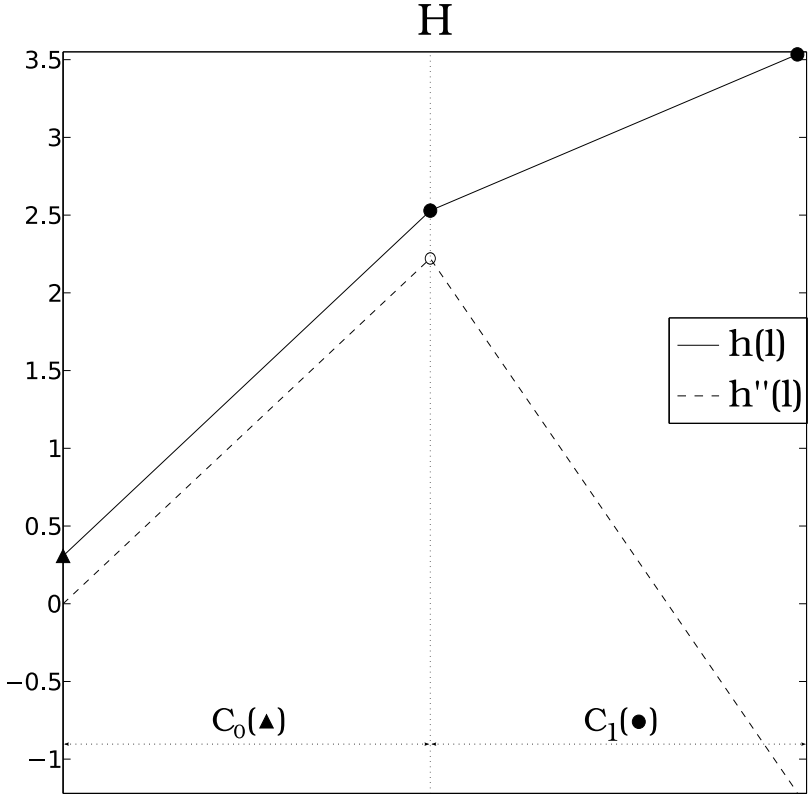


(b) Second step: cluster-based threshold

Figure 4.9: Checkpoint insertion for CESGA CalcuNetw



(a) First step: shape-based threshold



(b) Second step: cluster-based threshold

Figure 4.10: Checkpoint insertion for CESGA Fekete

Table 4.11: Detail of loops selected by the shape-based threshold for CESGA Fekete

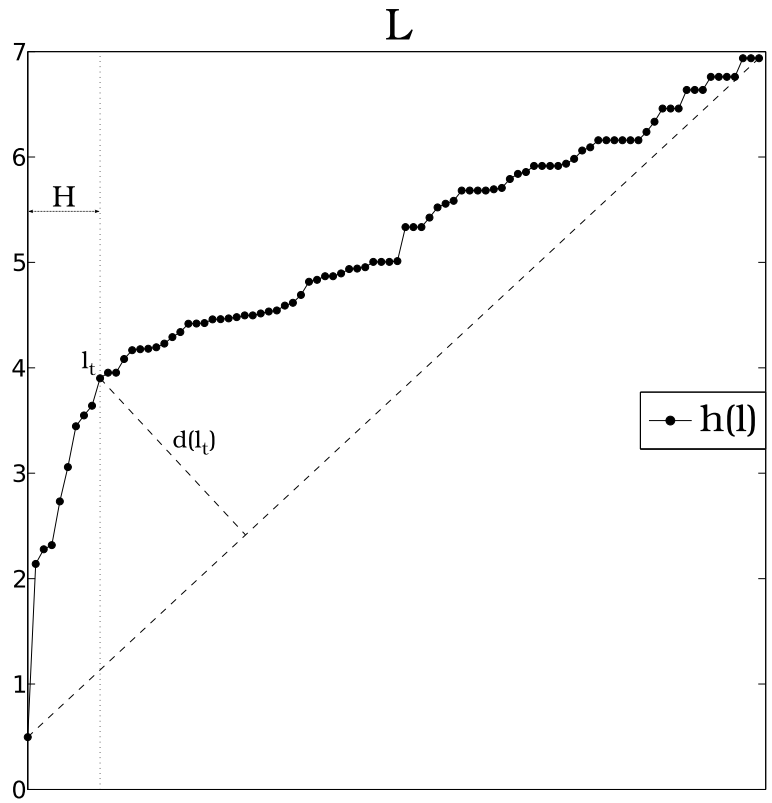
Loop #	**Statements**	**Accesses**	$h(l)$
1	77	181	0.3077
2	9	9	2.5288
3	2	4	3.5342
Total program statements			119
Total program accesses			238

- DBEM

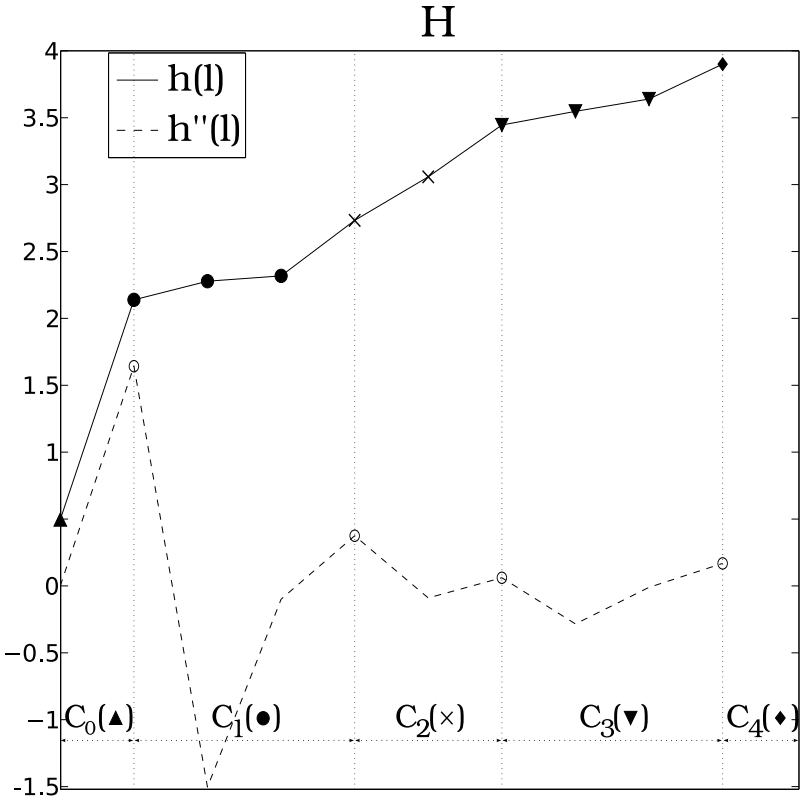
This is the biggest application in the experimental set, with a total of 92 loop nests being considered by the compiler. Out of them, 10 are selected by the shape-based threshold, as shown in Figure 4.11(a), and categorized into five different clusters by the second thresholding method detailed in Figure 4.11(b). Values for $h(l)$ in the H subset are detailed in Table 4.12. Loop #1 is responsible for the creation of the equation systems that are resolved later in loop #2, which is the main computational loop of the application. The manual analysis performed in the initial assessment of the application determined that both loops should be checkpointed. The automatic analysis decides to checkpoint both, and also two more loops that are conservatively selected. Although loop #2 is considerably bigger than both of them, the huge size of loop #1 makes them comparable when in the context of the entire application. Together, C_0 and C_1 are responsible for 65.69% of the total variation of $h(l)$ in H.

- STEM-II

This application, the second biggest in the experimental set, has most of its code inside a single loop, which is the one manually selected for checkpointing during the initial tests and also the only one checkpointed by the automatic checkpoint insertion algorithm. Figures 4.12(a) and 4.12(b) show the first and second thresholding steps, respectively. As can be seen in Table 4.13, loop #1 is so big that it completely neglects the rest of the nests in the application, and is the only selected from the two belonging to the H subset.

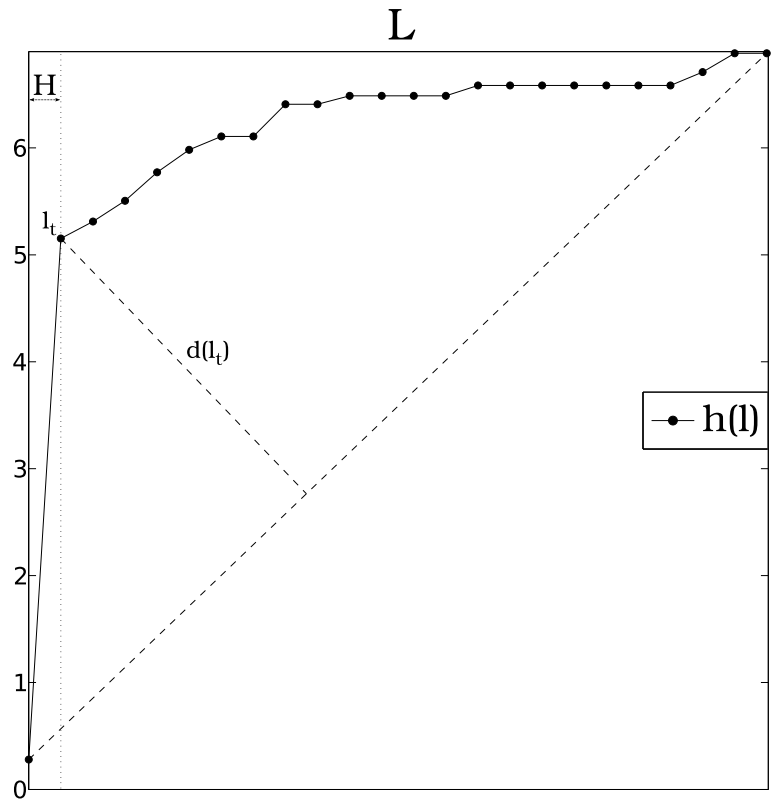


(a) First step: shape-based threshold

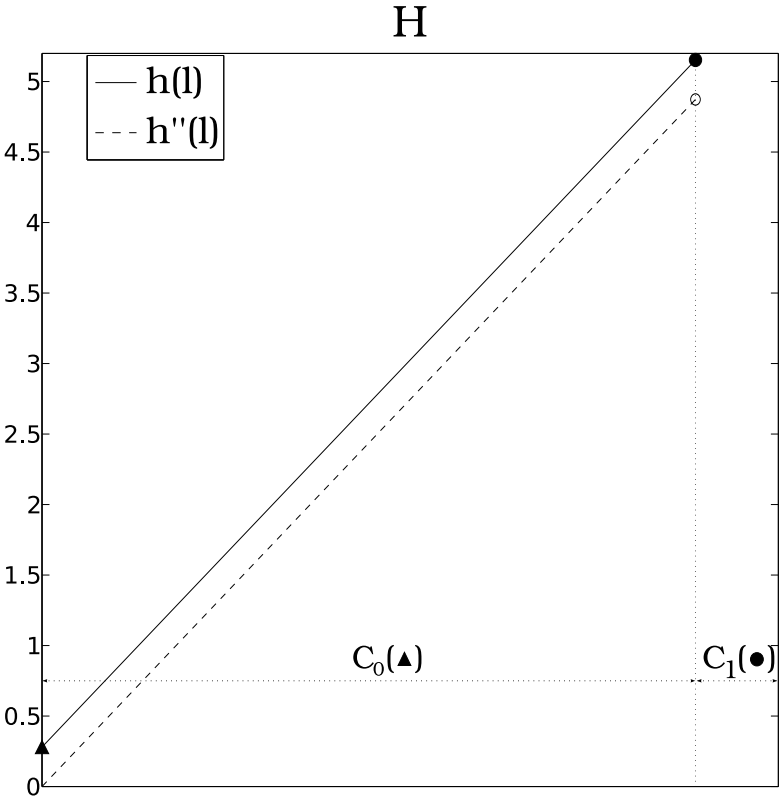


(b) Second step: cluster-based threshold

Figure 4.11: Checkpoint insertion for DBEM



(a) First step: shape-based threshold



(b) Second step: cluster-based threshold

Figure 4.12: Checkpoint insertion for STEM-II

Table 4.12: Detail of loops selected by the shape-based threshold for DBEM

Loop #	**Statements**	**Accesses**	$h(l)$
1	1924	2869	0.5017
2	333	438	2.0797
3	214	426	2.2838
4	182	457	2.3236
5	135	237	2.7385
6	84	180	3.0640
7	89	66	3.4747
8	52	95	3.5499
9	60	69	3.6266
10	53	41	3.9065
Total program statements			3330
Total program accesses			5262

Table 4.13: Detail of loops selected by the shape-based threshold for STEM-II

Loop #	**Statements**	**Accesses**	$h(l)$
1	1673	4305	0.3297
2	9	12	5.1538
Total program statements			2731
Total program accesses			5635

4.2.2. Compilation times

A summary of the characteristics of the test applications, including number of files, lines of code (LOCs), time it takes the CPPC compiler to instrument them, number of loop nests in the code, and number of checkpoints and variable registrations inserted by the CPPC compiler is shown in Table 4.14. Compilation times were measured in a desktop machine, with an Intel Core2 Duo CPU at 3.00 Ghz. and 1 GB of RAM. Although the CPPC compiler is a experimental tool and is not optimized for production use, it can be seen that compile times are acceptable for all test applications, and mostly dependent on the number of lines of code of the source. The higher compile time is obtained for the DBEM application, with 12533 LOCs, and is 77.14 seconds.

A more in-depth analysis of compilation times is given in Table 4.15, where they have been broken down into the times for each of the analyses described in Chapter 3. Times for code parsing, insertion of initialization and shutdown CPPC calls,

instrumentation of non-portable calls and open files, conversion to CPPC statements, data flow analysis, communication analysis, checkpoint insertion, language-dependent transforms and code generation are detailed. Although the number of applications is not nearly enough as to develop a complete mathematical model of execution times, some tendencies can be inferred from these times. Most of the analyses run in linear time with respect to the lines of code of the compiled application. Particularly, code parsing, the insertion of the CPPC initialization and shutdown calls, the analysis of non-portable calls, instrumentation of open files, transformation to CPPC statements, and language-dependent transforms appear to be $O(n)$, being n the number of LOCs in the code. The data flow analyses are $O(n)$ as well, but very dependent on the programming language the application is coded in. If a linear regression model were fitted to the times for the data flow analysis, two different models would have to be created for the analysis of C and Fortran 77 applications. The communications analysis tends to $O(n^2)$. The checkpoint insertion analysis does not depend on the LOCs of the application, but rather on the number of loop nests ($\#L$), being $O(\#L^2)$. As for the times described as "code generation", they include many compilation passes, such as insertion of the actual registration calls for restart-relevant variables, control flow code, etc. Thus, it cannot be analyzed precisely as a whole, but should be broken down in each of the individual analysis that are performed. However, as a general pattern, it may be characterized as $O(n^2)$.

4.3. Library

Runtime tests of the CPPC library were performed on the Finis Terrae supercomputer hosted by CESGA (Galician Supercomputing Center). It consists of 144 computation nodes:

- 142 HP Integrity rx7640 nodes with 16 Itanium Montvale cores and 128 GB of RAM per node.
- 1 HP Integrity Superdome node, with 128 Itanium Montvale cores and 1024 GB of RAM.
- 1 HP Integrity Superdome node, with 128 Itanium 2 cores and 384 GB of RAM.

Table 4.14: Statistics and compilation times for test applications

Application		Files	LOCs	Compile time	Loop nests	#Chkpts.	#Registers
NAS NPB-MPI v3.1	BT	18	3650	14.19 s.	25	1	121
	CG	1	1044	2.61 s.	13	2	38
	EP	1	180	0.90 s.	4	1	14
	FT	1	1269	4.82 s.	20	1	31
	IS	1	672	3.88 s.	6	3	34
	LU	25	3086	7.78 s.	35	1	74
	MG	1	1618	13.08 s.	12	1	34
	SP	24	3148	13.69 s.	25	4	143
CESGA	CalcuNetw	5	810	57.34 s.	14	1	28
	Fekete	1	182	0.86 s.	6	1	17
	DBEM	42	12533	77.14 s.	92	4	279
	STEM	110	6506	15.14 s.	24	1	94

Table 4.15: Breakdown of compilation times for test applications

Application		Total time	Parsing	Init+Shutdown	Non-portable	Open files	CPPC stmts.
NAS NPB-MPI v3.1	BT	14.19 s.	2974 ms.	346 ms.	106 ms.	110 ms.	445 ms.
	CG	2.61 s.	415 ms.	80 ms.	15 ms.	17 ms.	84 ms.
	EP	0,9 s.	185 ms.	55 ms.	6 ms.	6 ms.	14 ms.
	FT	4.82 s.	992 ms.	159 ms.	43 ms.	48 ms.	264 ms.
	IS	3.88 s.	395 ms.	77 ms.	22 ms.	23 ms.	22 ms.
	LU	7.78 s.	1564 ms.	203 ms.	57 ms.	61 ms.	282 ms.
	MG	13.08 s.	1278 ms.	217 ms.	60 ms.	64 ms.	332 ms.
	SP	13.69 s.	1815 ms.	199 ms.	57 ms.	60 ms.	230 ms.
CESGA	Calcunetw	24.55 s.	906 ms.	110 ms.	96 ms.	104 ms.	130 ms.
	Fekete	0.86 s.	189 ms.	54 ms.	4 ms.	6 ms.	12 ms.
	DBEM	77.14 s.	4975 ms.	344 ms.	117 ms.	144 ms.	443 ms.
	STEM-II	15.14 s.	1786 ms.	220 ms.	65 ms.	82 ms.	195 ms.

Application		Data flow	Comms.	Chkpt. insertion	Language	Code generation
NAS NPB-MPI v3.1	BT	3170 ms.	2449 ms.	851 ms.	100 ms.	2851 ms.
	CG	519 ms.	636 ms.	80 ms.	18 ms.	515 ms.
	EP	123 ms.	78 ms.	11 ms.	4 ms.	229 ms.
	FT	1180 ms.	837 ms.	273 ms.	46 ms.	952 ms.
	IS	1043 ms.	1679 ms.	48 ms.	19 ms.	337 ms.
	LU	1920 ms.	1304 ms.	388 ms.	55 ms.	1565 ms.
	MG	1561 ms.	7707 ms.	317 ms.	60 ms.	1349 ms.
	SP	2211 ms.	1756 ms.	783 ms.	52 ms.	6239 ms.
CESGA	Calcunetw	21955 ms.	0 ms.	198 ms.	84 ms.	999 ms.
	Fekete	146 ms.	70 ms.	15 ms.	4 ms.	176 ms.
	DBEM	17198 ms.	15233 ms.	20334 ms.	101 ms.	18267 ms.
	STEM-II	5946 ms.	3553 ms.	1448 ms.	48 ms.	1387 ms.

Table 4.16: Number of nodes and cores used for runtime tests for each application

Application		Nodes	Cores per node	Total cores
NAS NPB-MPI v3.1	BT	3	12	36
	CG	2	16	32
	EP			
	FT			
	IS			
	LU			
	MG			
	SP	3	12	36
CESGA	CalcuNetw	1	1	1
	Fekete	2	16	32
	DBEM			
	STEM-II			

The nodes are conected through an Infiniband 4x DDR network with a bandwith of 20 Gbps. The applications were executed on the HP Integrity rx7640 nodes. Table 4.16 details the number of nodes used for each application in runtime tests. Whenever possible, two nodes with 16 cores each were used for the execution. However, the NAS BT and SP applications require the number of parallel processes to be a square, which caused the number of processes to be raised to 36, allocating 3 nodes and using 12 cores of each of them. The remaining 4 cores in each node were allocated to prevent applications belonging to other users from being executed on them, thus disrupting the results. CESGA CalcuNetw is a sequential application. A whole node was allocated for its execution, while only one of its cores was used for the tests.

Executed measurements include: generated state file sizes, time for state file generation, checkpointing overhead and restart times. For proving portability, cross-restarts were executed on the Finis Terrae using state files generated by the Muxia cluster, hosted by the Computer Architecture Group of the University of A Coruña, consisting of Intel Xeon 1.8 Ghz nodes, with different compilers and MPI implementations than those found in the Finis Terrae.

The NAS CG, IS and SP, and DBEM applications create more than one checkpoint file during their execution. In order to provide normalized results for the state file sizes, state file creation time and restart time tests, these parameters were measured for the biggest checkpoints created in each case. This makes it easier to provide graphical comparisons of the results for different applications.

4.3.1. State file sizes

When using CPPC's spatially coordinated technique, the incurred overhead will only depend on the overhead introduced by the checkpoint file dumping. This overhead heavily depends on the size of the data to be dumped. Thus, the first parameter to be measured is how the variable level approach affects checkpoint file sizes. In order to analyze this effect, state file sizes have been measured for different checkpointing configurations, and compared to sizes obtained using a full checkpointer. For most applications, file sizes also vary depending on the number of processes involved in the execution, because the sizes of array data are modified to match the problem size. The exceptions are NAS EP, CESGA Fekete, DBEM and STEM-II, which statically allocate a fixed array size that determines the maximum allowed problem size. The results of this test are shown in Figures 4.13 to 4.23. The values tagged as "Automatic" are file sizes obtained by the automatic variable registration included in the compiler. "Compressed" shows file sizes using the HDF-5 writer with compression enabled. Compression is only applied to vector variables (pointers and arrays) larger than a certain user-specified limit, which in this case was set to 2000 elements. For comparison purposes, "Optimal" shows the optimal file sizes obtained by a manual analysis, and "Full data" presents the sizes obtained for a checkpointer that stores all the application data. Figure 4.24 shows a summary of the sizes obtained for all applications using the number of processes used for runtime tests shown in Table 4.16, and includes the CESGA CalcuNetw application, which does not have a figure of its own because it is a sequential application.

Sizes obtained using automatic analyses are close to the optimal ones, for most applications. The difference between both is due to the registration of unnecessary array sections because of the conservative approach of the compiler, as explained in Section 3.2.5. Variable level checkpointing usually achieves very important size reductions when compared to full data sizes. Table 4.17 gives the exact details on reduction percentages for the number of processes used in runtime tests for each application. It includes the size of the files created by the automatic live variable analysis, the reduction obtained with respect to the size of the files created by a full data checkpointer, and the optimal reduction (bracketed).

The high compression rates obtained for DBEM and STEM-II (97.86% and 96.10%, respectively) are due to the fact that these applications statically allocate arrays which are oversized to fit a maximum problem size. This is directly related with the fact that checkpoint sizes for these applications do not vary with the num-

Table 4.17: Performance of the automatic variable registration algorithm

	Application	Automatic size (MB)	Reduction (Optimal)
NAS NPB-MPI v3.1	BT	17.30	93.77% (98.55%)
	CG	19.77	93.55% (93.64%)
	EP	1.04	88.87% (99.71%)
	FT	40.10	87.32% (92.38%)
	IS	72.1	78.28% (92.76%)
	LU	8.26	51.81% (64.26%)
	MG	15.64	60.49% (63.15%)
	SP	19.77	86.05% (91.99%)
CESGA	Fekete	1.60	82.59% (92.81%)
	CalcuNetw	3.12	73.78% (73.80%)
	DBEM	275.67	5.94% (10.70%)
	STEM-II	114.13	39.72% (83.42%)

ber of processes executing them. Rather, the more processes, the bigger the problem that can be solved. As a result, an important amount of empty memory is allocated in our tests, resulting in high compression rates.

4.3.2. State file creation time

The Finis Terrae machine exhibits a high variability between times obtained by different executions of the same experiment. Allocating entire nodes reduces the variability, but does not entirely solve the problem. Thus, performing a large number of experiments and statistically analyzing the results is a better approach than just giving the minimum, maximum, or mean times for each experiment. The experimental approach consisted on measuring the creation time for the bigger state file for each application a minimum of 500 times, followed by the elimination of outliers and the calculation of the 99% confidence interval for the expected creation times. The approach used for outlier identification was to discard observations bigger than a certain threshold. This threshold is defined for each application as the third quartile of the data series plus 1.5 times the interquartile range (IQR). This is a classical approach to outlier identification [68]. Table 4.18 shows the 99% confidence interval for the mean creation times of a standard HDF-5 state file, the same HDF-5 file including a CRC-32 error detection scheme, and for compressed HDF-5 files for all applications. Also, the minimum obtained times are shown for comparison purposes. Note that these times correspond to the raw dumping time

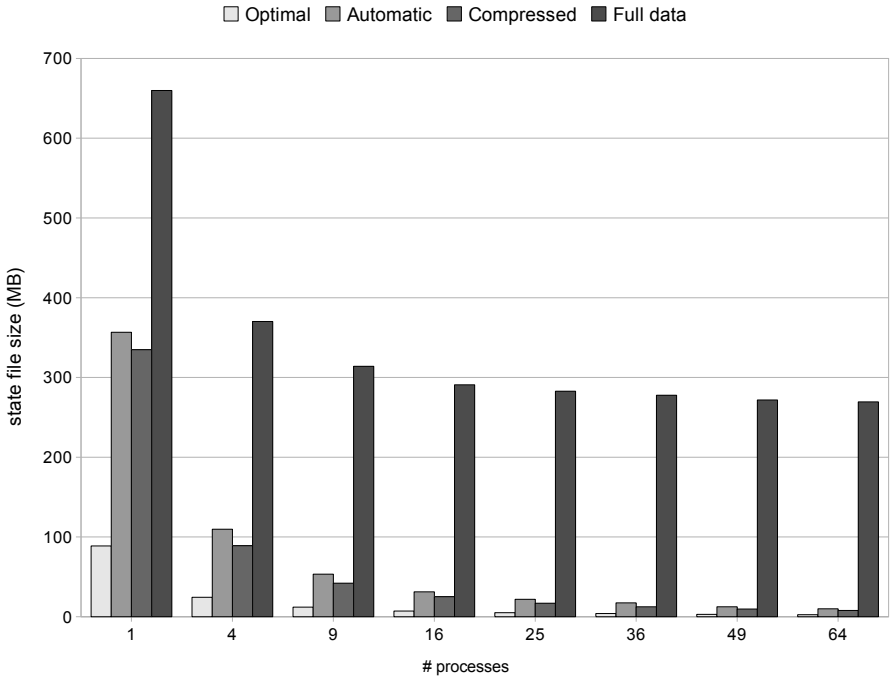


Figure 4.13: File sizes for NAS BT

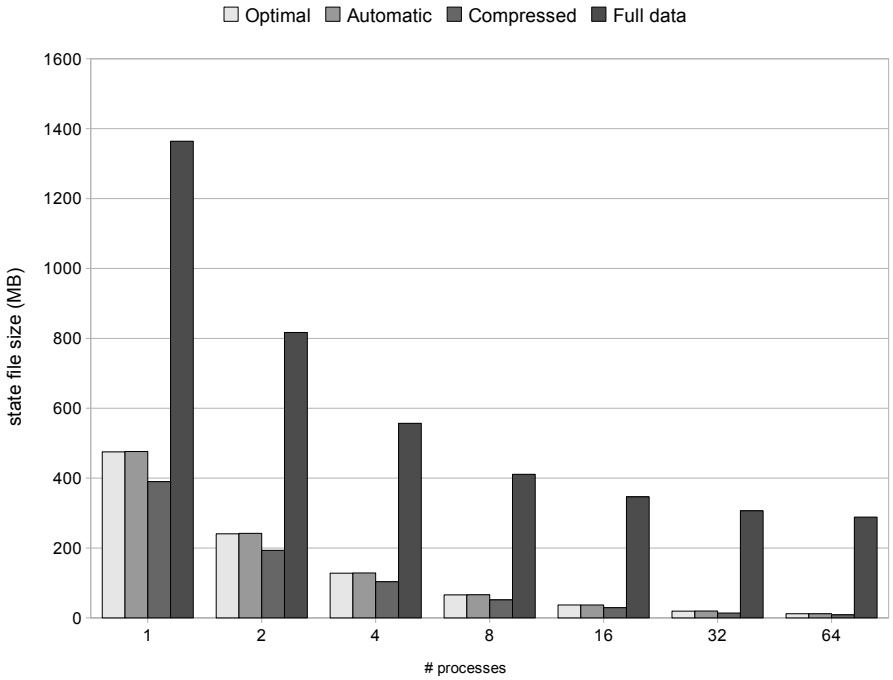


Figure 4.14: File sizes for NAS CG

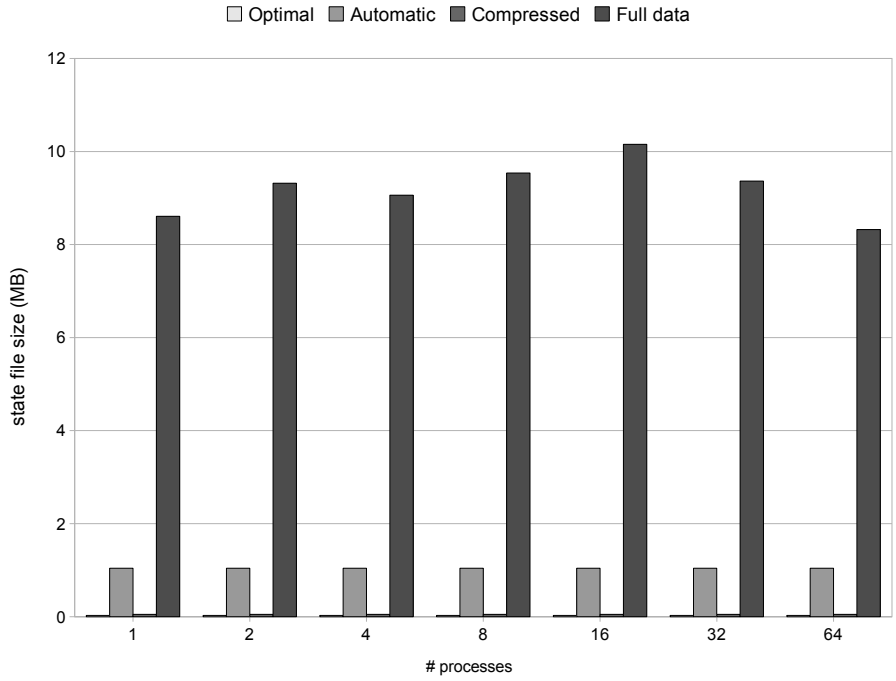


Figure 4.15: File sizes for NAS EP

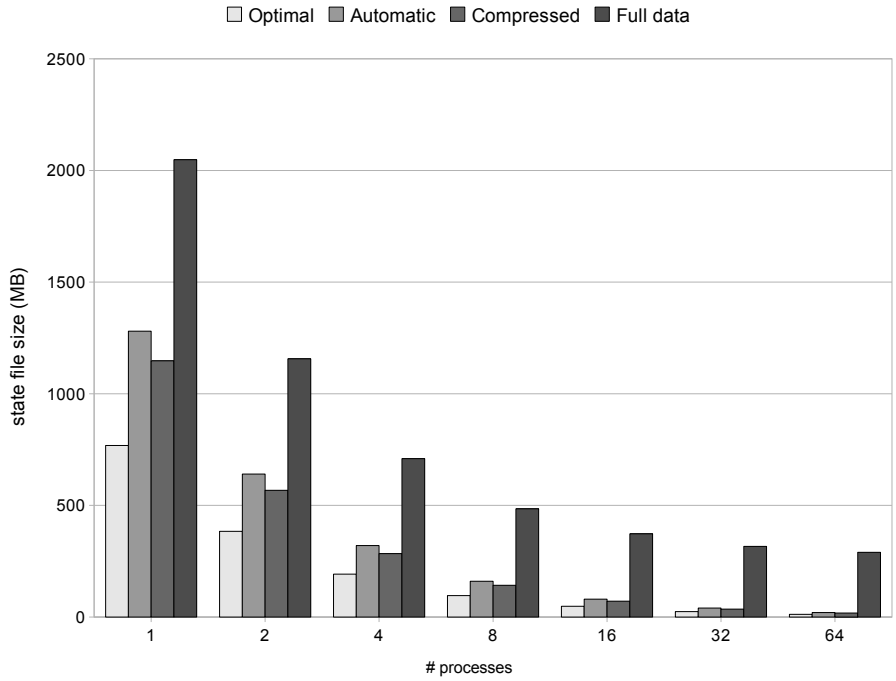


Figure 4.16: File sizes for NAS FT

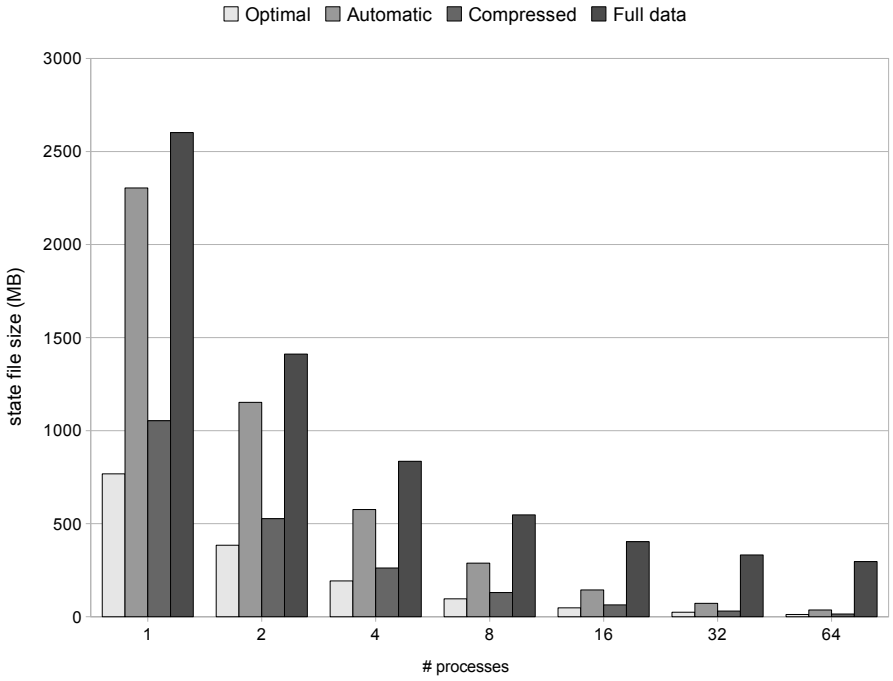


Figure 4.17: File sizes for NAS IS

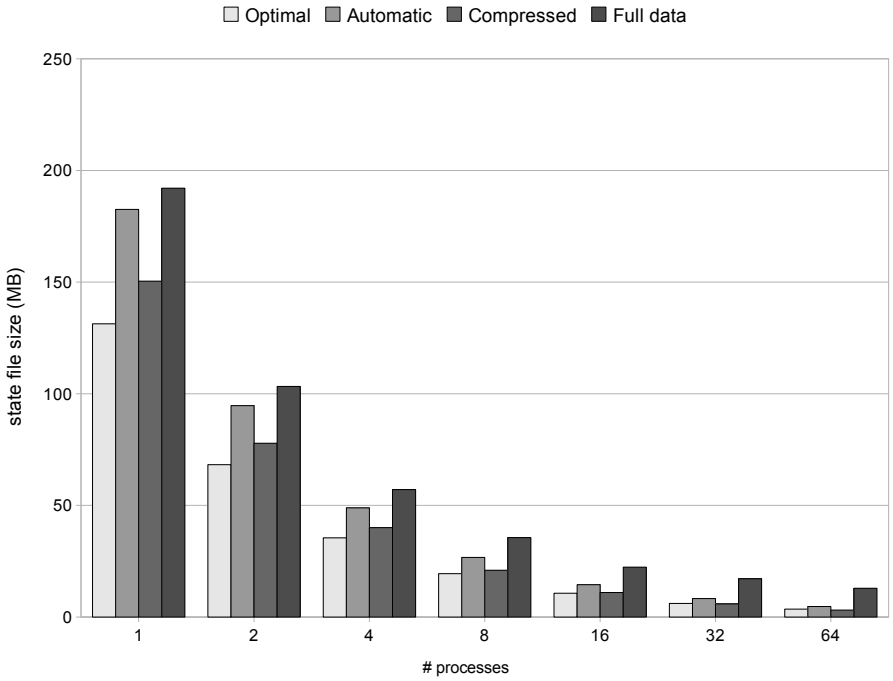


Figure 4.18: File sizes for NAS LU

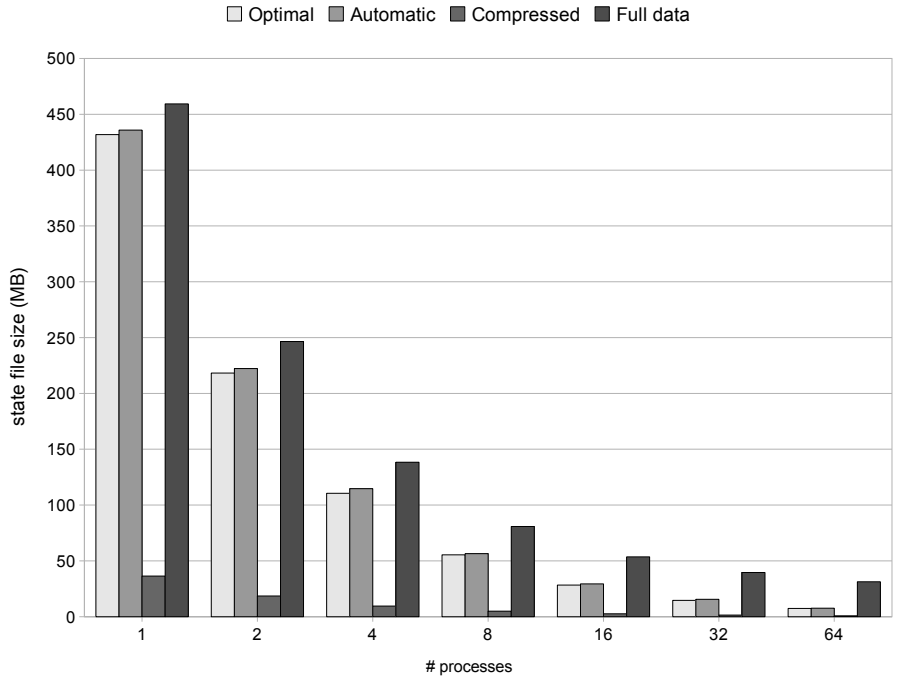


Figure 4.19: File sizes for NAS MG

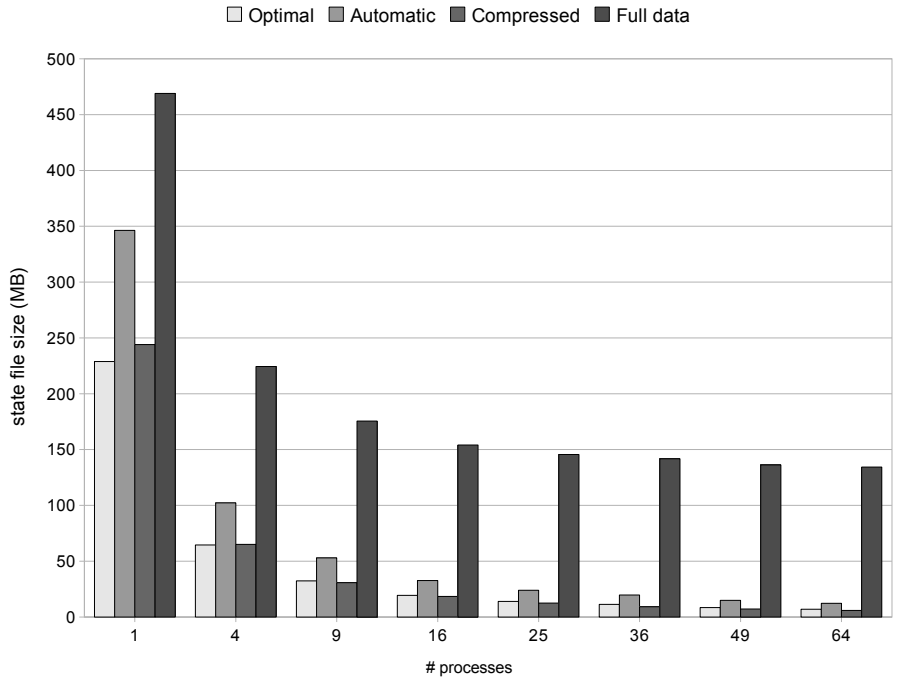


Figure 4.20: File sizes for NAS SP

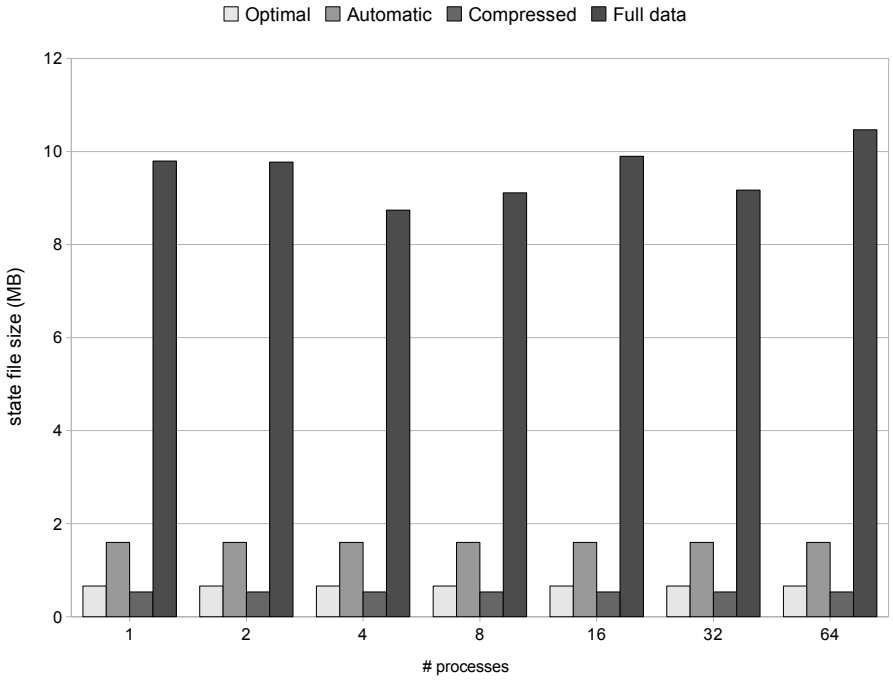


Figure 4.21: File sizes for CESGA Fekete

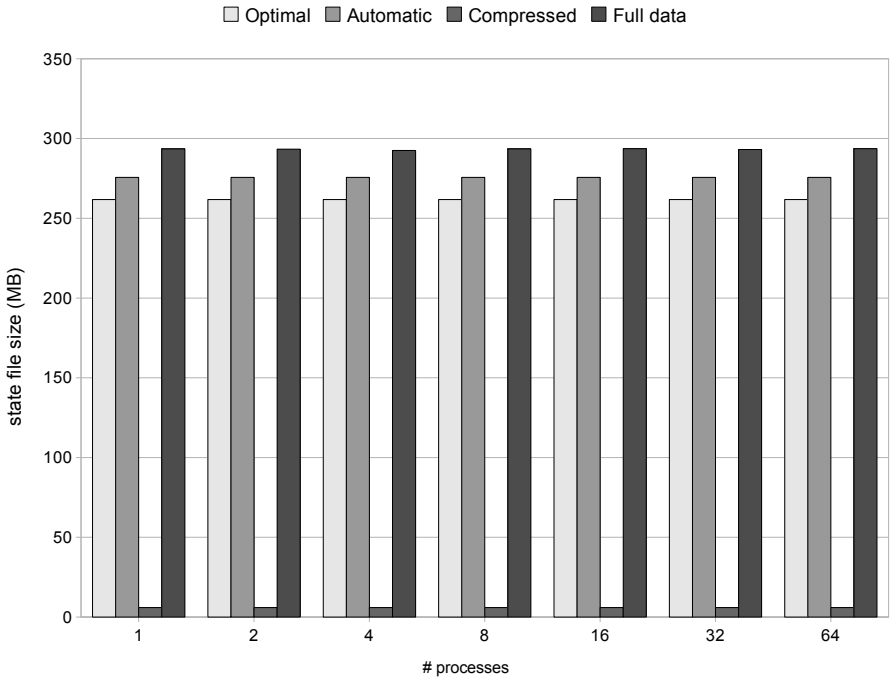


Figure 4.22: File sizes for DBEM

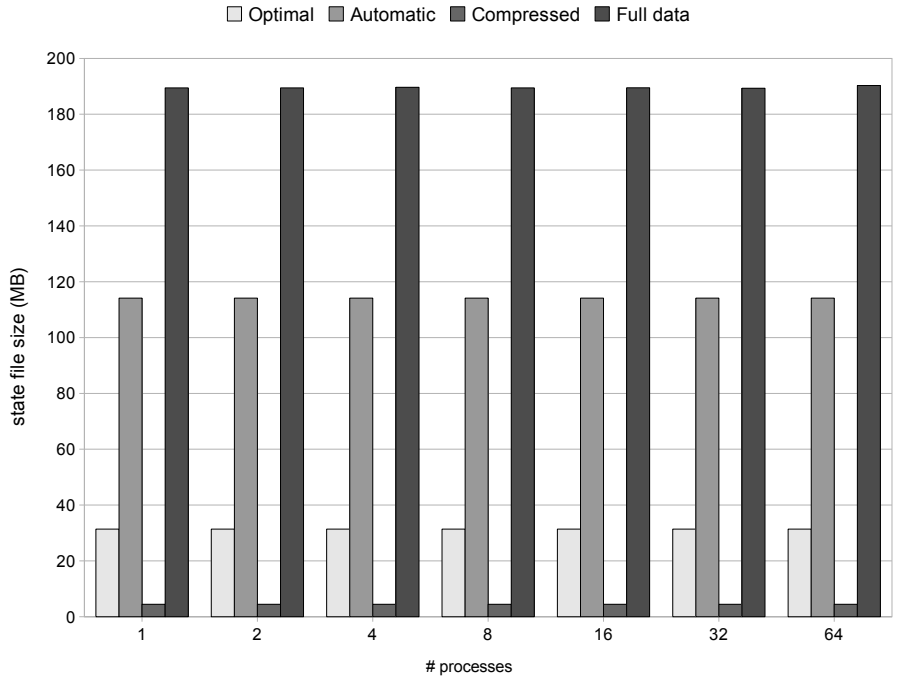


Figure 4.23: File sizes for STEM-II

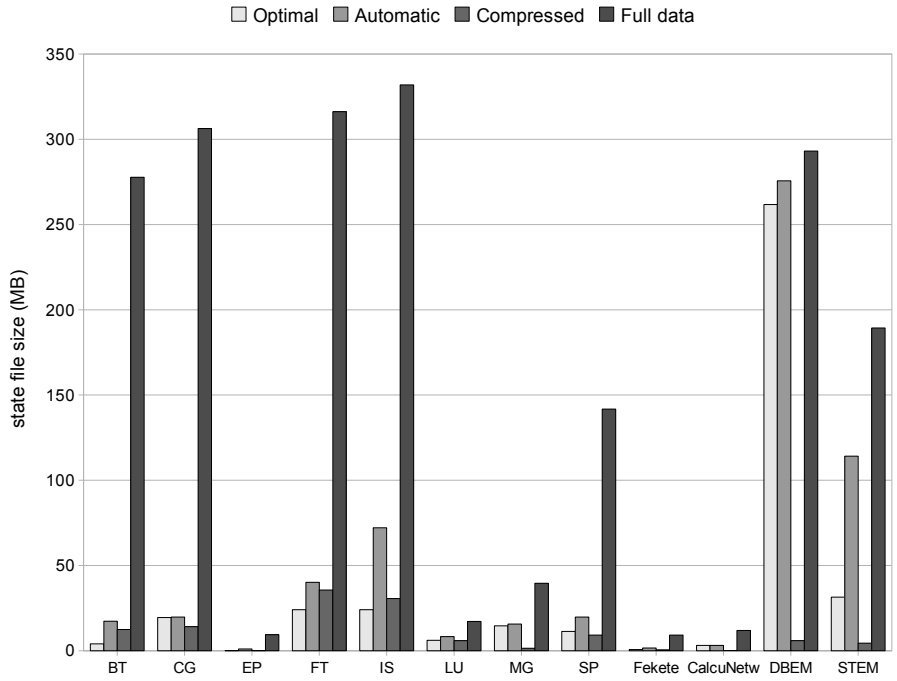


Figure 4.24: Summary of file sizes

of a single checkpoint, not the real contribution of dumping times to the checkpoint overhead, which is reduced by using multithreaded dumping.

Written data are tagged by the HDF-5 library to allow for conversions, if needed, when restarting the application. This improves checkpoint mode performance, moving the conversion overhead to the restart mode, which should be a much less frequent operation. A regression analysis on the dumping times with respect to the checkpoint file sizes yields a linear relationship between both factors in both the standard and the CRC-32 file creations. Using compression, however, the relationship is not linear, since the dumping time depends on the number of variables to be compressed and the entropy of the contained data. As can be seen, compressing data heavily increases overall dumping times. Therefore, it should be enabled only when the physical size of state files is critical: for instance, if there are problems with disk quotas or when the files are going to be transferred using a slow network. Also, it should be noted that the interval obtained for the mean creation times of compressed files for the NAS MG application is particularly large (about 0.25 seconds). This is due to the fact that the entropy of the stored data does not remain constant through the execution, causing compression times to increase as the execution progresses, and a high variance of the obtained times.

Figure 4.25 graphically compares the results shown in the table above, using the maximum estimated value (i.e. the upper end of the confidence intervals) for each application.

4.3.3. Checkpoint overhead

To reduce the overhead introduced by file generation, multihreaded state dumping has been implemented. When performing the overhead tests, the problem of the experimental variability in Finis Terrae has to be carefully dealt with. The load of the allocated nodes dramatically impacts execution times. To handle this situation, full nodes were reserved for the execution of the experimental applications as explained before. Moreover, it is important that the applications are executed on the same nodes and in the same conditions. Thus, once the nodes were allocated, a sequence of N experiments was run on them randomly selecting the type of each of them before being executed. Thus, for instance, if 500 experiments were scheduled, before starting each of them a random number was generated, and depending on it being odd or even the original version of the application or the CPPC-instrumented one was run. For the shorter applications, the number N of experiments was limited

Table 4.18: Checkpoint file creation times (seconds)

	Application	STANDARD			CRC-32			COMPRESSED		
		$\bar{x}_{min}$	$\bar{x}_{max}$	min	$\bar{x}_{min}$	$\bar{x}_{max}$	min	$\bar{x}_{min}$	$\bar{x}_{max}$	min
NAS NPB-MPI v3.1	BT	0.1374	0.1436	0.1162	0.2221	0.2286	0.2016	1.3248	1.3299	1.3095
	CG	0.0904	0.0991	0.0896	0.1914	0.1988	0.1728	1.7216	1.7256	1.7093
	EP	0.0065	0.0066	0.0061	0.0112	0.0113	0.0108	0.1195	0.1198	0.01187
	FT	0.1612	0.1717	0.1259	0.3601	0.3685	0.3217	3.4236	3.4339	3.3957
	IS	0.2875	0.2996	0.2457	0.6500	0.6599	0.6008	6.2744	6.3449	6.1869
	LU	0.0776	0.0810	0.0661	0.1145	0.1172	0.1040	0.6191	0.6279	0.6006
	MG	0.1035	0.1077	0.0890	0.1796	0.1840	0.1666	0.6795	0.9111	0.5113
	SP	0.1633	0.1692	0.1471	0.2657	0.2706	0.2509	1.2629	1.2699	1.2387
CESGA	Fekete	0.0251	0.0275	0.0187	0.0309	0.0322	0.0267	0.0818	0.0824	0.0802
	CalcuNetw	0.0207	0.0214	0.0189	0.0353	0.0363	0.0313	0.0964	0.0968	0.0951
	DBEM	2.2384	2.2725	2.1339	3.6321	3.6769	3.4345	7.0284	7.0646	6.8588
	STEM-II	0.9898	1.0147	0.8724	1.5406	1.5617	1.4581	4.3067	4.3527	4.0857

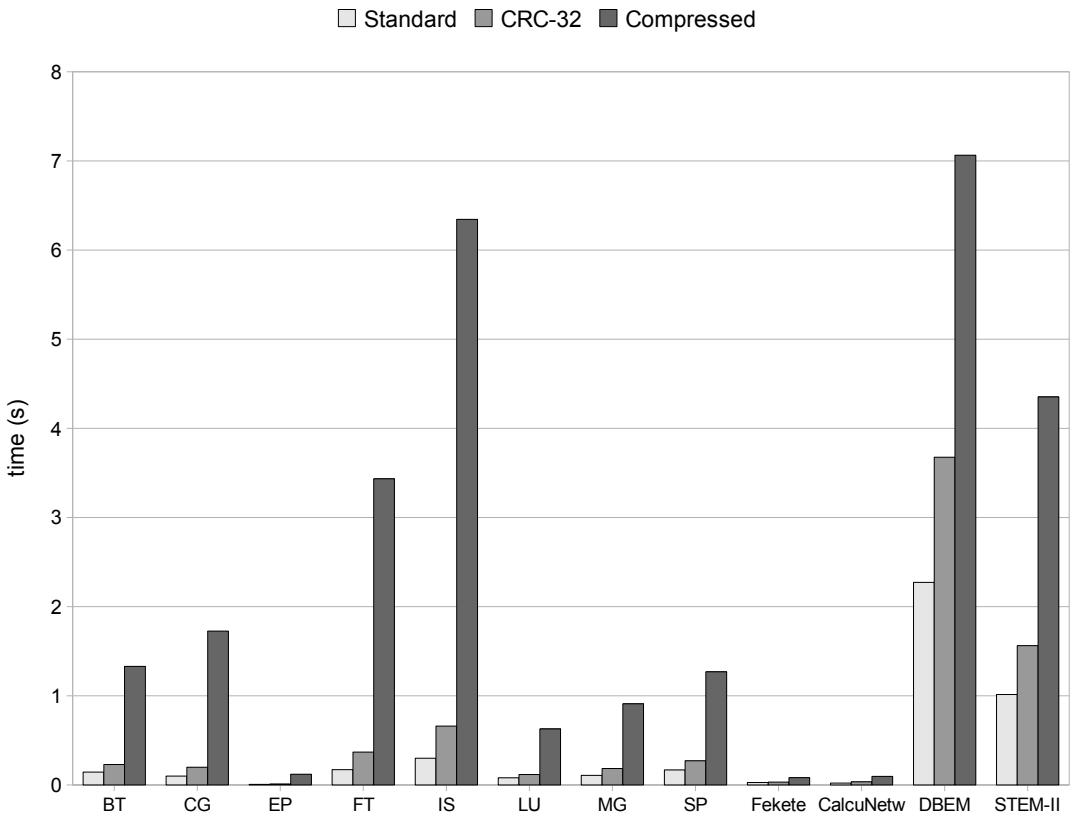


Figure 4.25: Maximum mean dumping times for test applications

to 500, a number which should provide statistically representative results without the need for more repetitions. For the larger ones this number was determined by the maximum allocation time of the nodes, which was 10 hours for the number of allocated cores. Performing more than one allocation is not a valid approach, since usually the obtained time series may not be assumed to come from the same statistical distribution. This experimental setup is designed to ensure that the variability exhibited by the machine affects all types of experiments to the same degree. Outliers are identified, as in the previous experiment, by removing observations bigger than the third quartile plus 1.5 times IQR in each of the series. Table 4.19 shows the number of regular and CPPC runs performed for each application, a 99% confidence interval for the checkpointing overhead, the minimum execution time for both the regular and CPPC versions of the code, and the maximum overhead percentage calculated as the upper limit of the overhead confidence interval divided by the minimum original runtime. As can be seen, sometimes the minimum execution

time for the CPPC version is below the minimum time for the original version of the application. This evidences the need for statistical analyses of execution times. Note that, even when considering the maximum possible overhead at 99% confidence and comparing it to the minimum time obtained for the execution of the original code, the overhead percentages remain low, usually in the 1% range, except for the IS application, which runs for about 25 seconds and therefore the 9.37% overhead obtained does not account for more than 2.5 seconds. These overheads include, besides file generation, all CPPC instrumentation (e.g. variable and parameter registration). Files were generated using the HDF-5 writer, with CRC-32 and without compression. One state file per checkpoint was generated for all the applications in the table.

The previously detailed experimental setup is not viable for the long-running applications, DBEM and STEM-II, since they run for more than 6 and 23 hours respectively. In order to obtain statistically significant measurements for these applications in the same way as for the shorter ones would require years of computation time. Instead, these applications were run in the Muxia cluster, owned by the Computer Architecture Group of the University of A Coruña, formed by Intel Xeon 1.8 Ghz nodes with 1 GB of RAM, connected through an SCI network. This cluster does not exhibit the high loads nor the high variability experienced in the Finis Terrae, and therefore makes it possible to reach plausible conclusions without the need for a large number of experiments. Thus, 10 executions of each version of the codes were run. Table 4.20 details the minimum execution times for the original versions of both DBEM and STEM-II executed on 4 cores, the checkpoint overhead incurred in seconds, and the overhead percentage. The checkpointing frequency for these applications was adjusted to approximately one checkpoint per hour, meaning that 23 checkpoints were created during the DBEM executions, and 7 checkpoints during the STEM-II ones.

4.3.4. Restart times

If a failure occurs, the restart time overhead must be taken into account in the global execution time. Restart times have been measured and split into its three fundamental phases explained in Chapter 2: negotiation, file read and effective data recovery. The negotiation process is measured from the `CPPC_Init_configuration()` call and until a global agreement about the restart position has been reached, and each process has identified the state file it must use for restarting. File read time

Table 4.19: Runtime overhead caused by checkpointing

	App.	Runs	$\bar{x}_{min}$	$\bar{x}_{max}$	Min. time	Min. CPPC time	Max. overhead
NAS NPB-MPI v3.1	BT	157 – 138	-2.8233	3.6010	562.87 s.	563.43 s.	0.6398%
	CG	73 – 74	-1.1995	1.2385	211.07 s.	210.25 s.	0.5868%
	EP	213 – 238	0.5300	0.8049	58.71 s.	58.32 s.	1.3710%
	FT	132 – 148	2.0734	3.6943	256.81 s.	259.14 s.	1.4385%
	IS	232 – 227	1.9252	2.4136	25.76 s.	27.66 s.	9.3696%
	LU	106 – 89	-2.7466	5.0750	857.30 s.	861.99 s.	0.5920%
	MG	249 – 218	-1.3841	0.9145	66.07 s.	66.87 s.	1.3841%
	SP	47 – 63	1.8426	3.1817	774.12 s.	775.48 s.	0.4110%
CESGA	Fekete	247 – 227	-0.1340	0.1874	51.63 s.	52.02 s.	0.3630%
	CalcuNetw	92 – 87	-3.5301	3.9988	351.71 s.	350.23 s.	1.1370%

Table 4.20: Runtime overhead on large-scale applications

Application	Min. time	Overhead	Overhead percentage
DBEM	80473.13 s.	256.02 s.	0.31%
STEM-II	21622.41 s.	101.14 s.	0.47%

measurements begin when the negotiation ends, and comprise all steps taken until the checkpoint data are loaded into memory and made available for the application to recover them. These include the identification of the writing plugin to be used, file opening and data reading. The recovery begins when the file read ends, and stops when CPPC determines that the restart process has finished, switching to checkpoint operation mode. This happens when the execution flow reaches the checkpoint statement where the file was generated in the original execution.

As with the measurements of checkpoint file creation times, 500 experiments were performed, outliers were discarded from each series, and a 99% confidence interval was calculated for each of the phases. The results are shown in Table 4.21. Figure 4.26(a) graphically compares the upper limit of the time intervals for each of the phases. As can be seen, the negotiation is the most costly phase. This is due to the fact that negotiation times have been measured for checkpoint files including CRC-32 codes. During the negotiation, application processes have to check the integrity of the files proposed for restart. If this experiment is performed with standard files without CRC-32 codes, negotiation times are reduced to the range of milliseconds for all applications. If performed on compressed files with CRC-32 codes, the time is reduced due to the files being smaller, thus requiring less time for the integrity checks to be performed.

The times obtained for the file read phase depend basically on the size of the files themselves, being linearly related. Read times would be increased when using compressed files because of the need for data decompression. They may also be increased when using state files generated on different architectures, due to the necessary format conversions. This effect is shown in Figure 4.26(b), which compares file read times in the Finis Terrae for both Finis Terrae- and Muxia cluster-generated state files. This test also serves to demonstrate portability, since restarts take place correctly using externally-generated files. Although the test was performed using the same statistical approach as above, only the upper limits of the intervals are shown for simplicity. The relationship between both times depends exclusively on the amount of data that needs to be converted, but the general trend is that the read time is nearly doubled when compared to the original one. The recovery times

Table 4.21: Restart times (seconds)

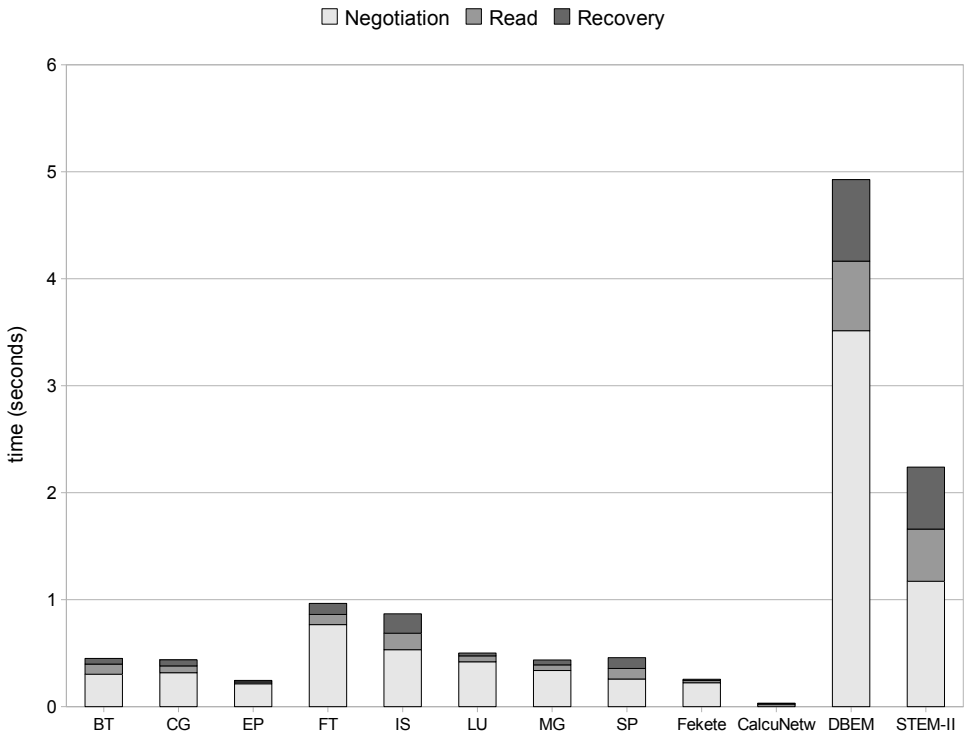
		NEGOTIATION		FILE READ		RECOVERY	
	Application	$\bar{x}_{min}$	$\bar{x}_{max}$	$\bar{x}_{min}$	$\bar{x}_{max}$	$\bar{x}_{min}$	$\bar{x}_{max}$
NAS NPB-MPI v3.1	BT	0.2668	0.3027	0.0913	0.0947	0.0507	0.0534
	CG	0.2845	0.3177	0.0608	0.0629	0.0551	0.0568
	EP	0.1714	0.2145	0.0152	0.0162	0.0132	0.0148
	FT	0.6850	0.7667	0.0911	0.0947	0.1010	0.1033
	IS	0.5248	0.5330	0.1476	0.1533	0.1783	0.1813
	LU	0.3675	0.4193	0.0535	0.0553	0.0252	0.0271
	MG	0.3057	0.3382	0.0500	0.0524	0.0440	0.0461
	SP	0.2454	0.2580	0.0937	0.0992	0.0903	0.1006
CESGA	Fekete	0.1775	0.2233	0.0198	0.0210	0.0100	0.0115
	CalcuNetw	0.0178	0.0180	0.0111	0.0113	0.0030	0.0030
	DBEM	3.3010	3.5141	0.6193	0.6498	0.7348	0.7624
	STEM-II	1.1269	1.1715	0.4711	0.4885	0.5468	0.5792

depend on the amount of data being recovered and the amount of code that must be re-executed in order to achieve a complete state recovery.

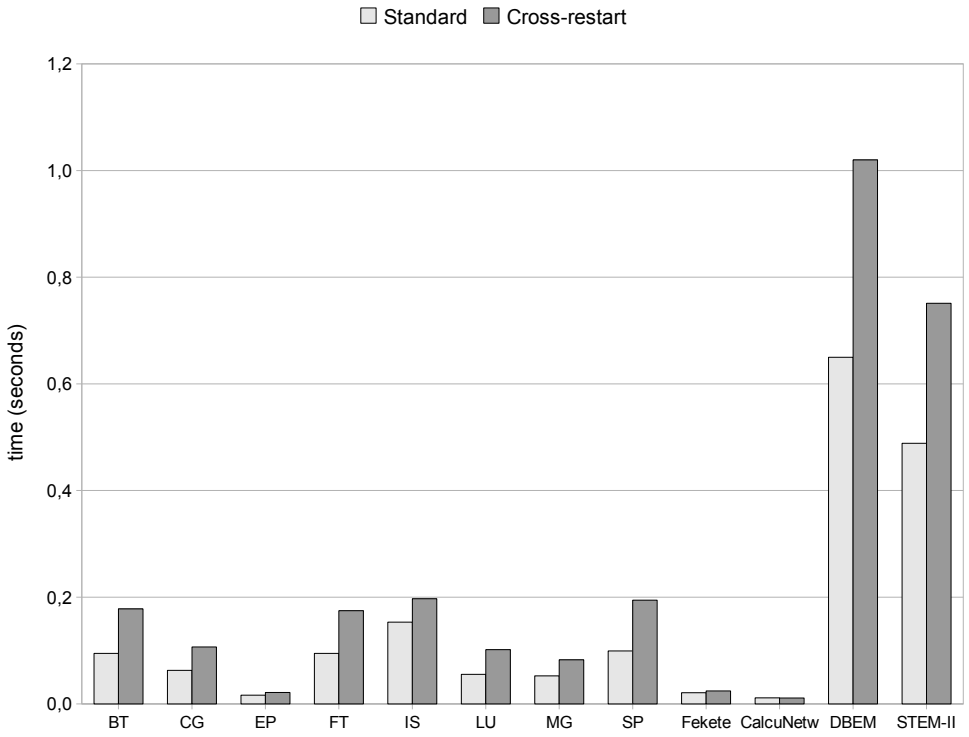
All these tests show restart times to be very low and fairly negligible in most situations. Restart times never exceed a second, except when working with large state files as is the case with the DBEM and STEM-II applications. Even in these cases, most of the restart overhead is introduced by the consistency checks on state files, while the restart protocol itself remains very light and negligible when compared to the total runtime of these applications.

4.4. Summary

This chapter gives a global overview of the performance obtained by the CPPC framework. A large number of very different applications have been used for experimental testing, showing adequate automatic processing of them by the framework in all cases. Results obtained by the compiler have been thoroughly described, and shown to be adequate in both accuracy of the results and compilation performance, particularly taking into account that the current implementation of the CPPC compiler has not been thoroughly optimized, neither implemented in a particularly efficient environment. After surveying the compilation results, the runtime behavior of the CPPC library has been addressed. File sizes have been analyzed as one of



(a) Upper limit of the confidence interval of restart times



(b) Effect of data conversions in file read times

Figure 4.26: Restart times for test applications

the fundamental factors in the global performance of the solution, and the benefits obtained by the use of the variable level approach have been addressed. The other key factor on checkpointing performance is the writing strategy used. To measure this impact, state file creation times were measured for all the writing strategies included in the CPPC package. In order to reduce the overhead introduced by the file creation step, CPPC may optionally use a multithreaded dumping algorithm. The global overhead introduced in the test applications when using this approach was also measured statistically to ensure the accuracy of the results. These have shown excellent checkpointing overheads, even when the presented numbers were the worst-case ones, and a much better behavior is to be expected in most executions. The overhead of the restart process has also been analyzed and shown to be negligible in reasonable execution scenarios (i.e. with standard failure rates). Cross-restart experiments have been executed, demonstrating low overheads for data conversions and the portability of the tool. All runtime tests, with the exception of the runs of large-scale applications, were performed on the Finis Terrae machine hosted by the Galician Supercomputing Center (CESGA). This supercomputer was ranked as #209 in the June 2008 Top 500 list [1]. It represents a typical machine in a supercomputing center, and performing the experimental evaluation in it provides a more accurate estimation of what can be expected of the CPPC framework in terms of performance. Also, it demonstrates that the framework may be used in a computing infrastructure on which the user has no superuser access or administration privileges.

Conclusions and future work

CPPC is a portable and transparent checkpointing infrastructure for long-running message-passing applications. It consists of a runtime library containing routines for checkpointing support, together with a compiler that automates the use of the library. The most remarkable contributions of this framework are:

- The compile-time coordination protocol: consistency issues are moved from runtime to both compile and restart time. At compile time, checkpoints are placed in safe points. At restart time, a negotiation between the processes involved in the execution of the application decides the safe point from which to restart. Process synchronization required by traditional coordinated checkpointing approaches is transferred to restart time, thus improving scalability. Moreover, this solution also enhances effiency, since both compiling and restarting an application are less frequent operations than checkpoint generation.

- The reduced state file sizes: CPPC works at variable level. It stores only those variables that are needed upon application restart. By restricting the amount of saved data, checkpoint files are made smaller and so checkpointing overhead decreases. This also improves the performance of network transfers, if necessary.

- The portable recovery of the application's state: state recovery is achieved in a portable way by means of both the hierarchical data format used for state dumping, and the selective re-execution of sections of non-portable code. This re-execution also provides scope for the checkpointing of applications linked to external libraries. Portability is a very interesting trait due to the inherent heterogeneity of current trends in high performance computing, such as Grid computing. CPPC-G [55] is an ongoing project developing an architec-

ture based on web services to manage the execution of CPPC fault tolerant applications on the Grid.

- The modular design of the library: the CPPC library is implemented in C++, using a highly modular design that allows for the flexible configuration of all its functionalities. It allows for dynamic configuration of file dumping formats, as well as selection of several capabilities such as file compression, critical when working with slow networks or limited amounts of disk space, or integrity checks. This highly modular design also allows for the use of the framework in environments where users do not have administrative rights, which is often the case in high performance computers. The use of the MVC pattern allows the use of the library from virtually any host programming language, through the implementation of thin software layers that adapt the application requests to the internal interface of the CPPC core.

- The fully transparent checkpointing: the CPPC compiler transforms the CPPC library into a totally transparent checkpointing tool, through the automation of all required code analyses and transformations. It includes a novel technique for automatic identification of safe points and computation-intensive loops based on the analysis of code complexity metrics, typical of software engineering approaches.

CPPC has been experimentally tested, demonstrating usability, scalability, efficiency and portability. It correctly performed application instrumentation and rollback-recovery for all the test cases, even using the same set of checkpoint files to perform restart on binary incompatible machines, with different C/Fortran compilers and MPI implementations. The experiments were performed on a publicly available computing infrastructure (the Finis Terrae supercomputer hosted by the CESGA), adopting statistical approaches to accurately provide performance estimations.

To our knowledge, CPPC is the only publicly available portable checkpointer for message-passing applications. CPPC is an open-source project, available at `http://cppc.des.udc.es` under GPL license.

This work has spawned the following publications:

- G. Rodríguez, M.J. Martín, P. González, J. Touriño y R. Doallo. Controlador/preCompilador de Checkpoints Portables. In *Actas de las XV Jornadas de Paralelismo*, pp. 253–258, Almería (Spain), September 2004.

- G. Rodríguez, M.J. Martín, P. González, J. Touriño y R. Doallo. On designing portable checkpointing tools for large-scale parallel applications. In *Proceedings of the 2nd International Conference on Computational Science and Engineering (ICCSE'05)*, pp. 191–203. Istanbul (Turkey), June 2005.

- G. Rodríguez, M.J. Martín, P. González, J. Touriño y R. Doallo. Portable Checkpointing of MPI applications. In *Proceedings of the 12th Workshop on Compilers for Parallel Computers (CPC'06)*, pp. 396–410, A Coruña (Spain), January 2006.

- G. Rodríguez, M.J. Martín, P. González y J. Touriño. Controller/Precompiler for Portable Checkpointing. *IEICE Transactions on Information and Systems*, E89-D(2):408–417, February 2006.

- G. Rodríguez, M.J. Martín, P. González, J. Touriño y R. Doallo. CPPC: Una herramienta portable para el checkpointing de aplicaciones paralelas. *Boletín de la red nacional de I+D, RedIRIS*, (80):57–71, April 2007.

- D. Díaz, X.C. Pardo, M.J. Martín, P. González y G. Rodríguez. CPPC-G: Fault-Tolerant Parallel Applications on the Grid. In *Proceedings of the 1st Iberian Grid Infrastructure Conference (IBERGRID'07)*, pp. 230–241, Santiago de Compostela (Spain), May 2007.

- G. Rodríguez, P. González, M.J. Martín y J. Touriño. Enhancing Fault-Tolerance of Large-Scale MPI Scientific Applications. In *Proceedings of the 9th International Conference on Parallel Computing Technologies (PaCT'07)*, Pereslavl-Zalessky (Russia). *Lecture Notes in Computer Science*, 4671:153–161, September 2007.

- D. Díaz, X.C. Pardo, M.J. Martín, P. González y G. Rodríguez. CPPC-G: Fault Tolerant Parallel Applications on the Grid. In 3rd Workshop on Large Scale Computations on Grids (LaSCoG'07), Gdańsk (Poland). *Lecture Notes in Computer Science*, 4967:852–859, May 2008.

- G. Rodríguez, X.C. Pardo, M.J. Martín, P. González, D. Díaz. A Fault Tolerance Solution for Sequential and MPI Applications on the Grid. *Scalable Computing: Practice and Experience*, 9(2):101–109, June 2008.

- G. Rodríguez, M.J. Martín, P. González, J. Touriño, R. Doallo. CPPC: A Compiler-Assisted Tool for Portable Checkpointing of Message-Passing Applications. In *Proceedings of the 1st International Workshop on Scalable Tools for*

High-End Computing (STHEC'08), held in conjunction with the 22nd ACM International Conference on Supercomputing (ICS'08), pp. 1–12, Kos (Greece), June 2008.

Future work

There are two main improvements that can be made to the CPPC framework. In the first place, the spatial coordination approach used has an important drawback: the need to specify checkpointing frequency in terms of the number of calls to the checkpointing function, instead of on a temporal basis. We are currently working on a lightweight, uncoordinated messaging protocol to allow processes to dynamically vary the checkpointing frequency based on a user-specified temporal frequency. This protocol involves performing all-to-all non-blocking communications regarding the speed at which each of the processes progresses through the application code, and adopting the necessary spatial frequency to ensure that the slowest process checkpoints at the user-specified rate.

The second improvement to be performed is related to the complexity analysis for checkpoint insertion. The current algorithm implemented in the CPPC compiler can be improved by using more complex metrics, in order to obtain a better approach to the optimal results.

Bibliography

[1] Top 500 supercomputer sites. http://www.top500.org. Last accessed September 2008.

[2] Fault tolerance under UNIX. *ACM Transactions on Computing*, 7(1):1–24, 1989.

[3] N. Aeronautics and S. Administration. The NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB. Last accessed September 2008.

[4] S. Agarwal, R. Garg, M. Gupta, and J. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS'04)*, pages 277–286, 2004.

[5] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Computing*, 6(3):227–236, 2003.

[6] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 1st edition, 2001.

[7] L. Alvisi. *Understanding the message logging paradigm for masking process crashes*. PhD thesis, Cornell University, Department of Computer Science, 1996.

[8] A. Baratloo, P. Dasgupta, and Z. Kedem. CALYPSO: A novel software system for fault-tolerant parallel processing on distributed platforms. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing (HPDC-4)*, pages 122–129, 1995.

[9] R. Batchu, J. Neelamegam, C. Zhenqian, M. Beddhu, A. Skjellum, Y. Dandass, and M. Apte. MPI/FT$^{\text{TM}}$: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In

Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid (CCGrid'01), pages 26–33, 2001.

[10] A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[11] B. Bhargava and S. Lian. Independent checkpointing and concurrent rollback for recovery – An optimistic approach. In *Proceedings of the 7th Symposium on Reliable Distributed Systems (SRDS'88)*, pages 3–12, 1988.

[12] A. Bouteiller, F. Capello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette. MPICH-V2: A fault-tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *Proceedings of the 15th ACM/IEEE Conference on Supercomputing (SC'03)*, pages 25–42, 2003.

[13] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of MPI programs. In *Proceedings of the 2003 ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 84–94, 2003.

[14] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. C^3: A system for automating application-level checkpointing of MPI programs. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pages 357–373, 2003.

[15] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Collective operations in application-level fault-tolerant MPI. In *Proceedings of the 17th International Conference on Supercomputing (ICS'03)*, pages 234–243, 2003.

[16] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1st edition, 1996.

[17] A. Carmona, A. Encinas, and J. Gesto. Estimation of Fekete points. *Journal of Computational Physics*, 225(2):2354–2376, 2007.

[18] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[19] Y. Chen, J. Plank, and K. Li. *CLIP: A checkpointing tool for message-passing parallel programs*, pages 182–200. The MIT Press, 2004.

[20] S.-E. Choi and S. Deitz. Compiler support for automatic checkpointing. In *Proceedings of the 16th International Symposium on High Performance Computing Systems and Applications (HPCS'02)*, pages 213–220, 2002.

[21] F. Cristian and F. Jahanian. A timestamp based checkpointing protocol for long-lived distributed computations. In *Proceedings of the 10th Symposium on Reliable Distributed Systems (SRDS'91)*, pages 12–20, 1991.

[22] E. Elnozahy, L. Alvisi, Y.-M. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[23] E. Elnozahy and J. Plank. Checkpointing for Peta-Scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, 2004.

[24] G. Fagg and J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *Lecture Notes in Computer Science*, 1908:346–353, 2000.

[25] S. Feldman and C. Brown. Igor: A system for program debugging via reversible execution. *ACM SIGPLAN Notices*, 24(1):112–123, 1989.

[26] R. Fernandes, K. Pingali, and P. Stodghill. Mobile MPI programs in computational Grids. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Computing (PPoPP'06)*, pages 22–31, 2006.

[27] N. C. for Supercomputing Applications. HDF-5: File Format Specification. http://hdf.ncsa.uiuc.edu/HDF5/doc/. Last accessed September 2008.

[28] E. Gamma, R. Helm, R. Jonhson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1st edition, 1994.

[29] R. Gioiosa, J. Sancho, S. Jiang, and F. Petrini. Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers. In *Proceedings of the 2005 ACM/IEEE Conference on High Performance Networking and Computing (SC'05)*, pages 9–23, 2005.

[30] P. González, T. Pena, and J. Cabaleiro. Dual BEM for crack growth analysis on distributed-memory multiprocessors. *Advances in Engineering Software*, 31(12):921–927, 2000.

[31] W. Gropp and E. Lusk. Fault Tolerance in Message Passing Interface Programs. *International Journal of High Performance Computing Applications*, 18(3):363–372, 2004.

[32] M. Hayden. The Ensemble system. Technical Report TR98-1662, Cornell University, Department of Computer Sciences, 1998.

[33] J. Hélary, A. Mostefaoui, and M. Raynal. Virtual precedence in asynchronous systems: concepts and applications. In *Proceedings of the 11th Workshop on Distributed Algorithms (WDAG'97)*, pages 170–184, 1997.

[34] D. Johnson and W. Zwaenepoel. Sender-based message logging. In *Proceedings of the 17th Annual International Symposium on Fault-Tolerant Computing (FTCS'87)*, pages 14–19, 1987.

[35] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.

[36] C. Landau. The checkpoint mechanism in KeyKOS. In *Proceedings of the 2nd International Workshop on Object Orientation on Operating Systems (IWOOOS'92)*, pages 86–91, 1992.

[37] S.-I. Lee, T. Johnson, and R. Eigenmann. Cetus – an extensible compiler infrastructure for source-to-source transformation. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pages 539–553, 2003.

[38] C.-C. Li, E. Stewart, and W. Fuchs. Compiler-assisted full checkpointing. *Software: Practice and Experience*, 24(10):871–886, 1994.

[39] C.-C. Li, E. Stewart, and W. Fuchs. Compiler-assisted full checkpointing. *Software: Practice and Experience*, 24(10):871–886, 1994.

[40] M. Litzkow, M. Livny, and M. Mutka. Condor – A hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS'88)*, pages 104–111, 1988.

[41] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report #1346, University of Wisconsin-Madison, 1997.

[42] M. Martín, D. Singh, J. Mouriño, F. Rivera, R. Doallo, and J. Bruguera. High performance air pollution modeling for a power plant environment. *Parallel computing*, 29(11–12):1763–1790, 2003.

[43] J. Mouriño, E. Estrada, and A. Gómez. CalcuNetw: Calculate measurements in complex networks. Technical Report 2005-003, Galician Supercomputing Center (CESGA), 2005.

[44] R. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, 1995.

[45] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, 1988.

[46] T. Parr. Antlr parser generator. http://www.antlr.org. Last accessed September 2008.

[47] J. Plank. *Efficient checkpointing on MIMD architectures*. PhD thesis, Princeton University, Department of Computer Science, 1993.

[48] J. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, 1995.

[49] J. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, 1995.

[50] J. Plank, J. Xu, and R. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, 1995.

[51] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogeneous architectures. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 58–67, 1997.

[52] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):221–232, 1975.

[53] S. Rao, L. Alvisi, and H. Vin. Egida: an extensible toolkit for low-overhead fault-tolerance. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS'99)*, pages 48–55, 1999.

[54] G. Rodríguez, M. Martín, P. González, and J. Touriño. Controller/Precompiler for Portable Checkpointing. *IEICE Transactions on Information and Systems*, E89-D(2):408–417, 2006.

[55] G. Rodríguez, X. Pardo, M. Martín, P. González, and D. Díaz. A fault tolerance solution for sequential and MPI applications on the Grid. *Scalable Computing: Practice and Experience*, 9(2):101–109, 2008.

[56] M. Russinovich and Z. Segall. Fault-tolerance for off-the-shelf applications and hardware. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS'95)*, pages 67–71, 1995.

[57] M. Sezgin and B. Sankur. Survey over image thresholding techniques and quantitative performance evaluation. *Journal of Electronic Imaging*, 13(1):146–165, 2004.

[58] D. Shires, L. Pollock, and S. Sprenkle. Program flow graph construction for static analysis of MPI programs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1847–1853, 1999.

[59] S. Siegel and G. Avrunin. Modeling wilcard-free MPI programs for verification. In *Proceedings of the 10th ACM Symposium on Principles and Practice of Parallel Computing (PPoPP'05)*, pages 95–106, 2005.

[60] S. Smale. Mathematical problems for the next century. *Mathematical Intelligencer*, 20(2):7–15, 1998.

[61] G. Stellner. Cocheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, pages 526–531, 1996.

[62] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[63] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 2000.

[64] V. Strumpen. Portable and fault-tolerant software systems. *IEEE Micro*, 18(5):22–32, 1998.

[65] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[66] Y. Tamir and C. Sequin. Error recovery in multicomputers using global checkpoints. In *Proceedings of the International Conference on Parallel Processing (ICPP'84)*, pages 32–41, 1984.

[67] S. Toueg and Ö. Babaoğlu. On the optimum checkpoint selection problem. *SIAM Journal on Computing*, 13(3):630–649, 1984.

[68] J. Tukey. *Exploratory data analysis.* Addison-Wesley, 1977.

[69] N. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions on Computers*, 46(8):942–947, 1997.

[70] Y.-M. Wang. *Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems.* PhD thesis, University of Illinois, Department of Computer Science, 1993.

[71] N. Woo, H. Jung, H. Yeom, T. Park, and H. Park. MPICH-GF: Transparent checkpointing and rollback-recovery for Grid-enabled MPI processes. *IEICE Transactions on Information and Systems*, E87-D(7):1820–1828, 2004.

[72] Xerces C++ Parser. http://xml.apache.org/xerces-c/. Last accessed September 2008.

[73] G. Zack, W. Rogers, and S. Latt. Automatic measurement of sister chromatid exchange frequency. *Journal of Histochemistry & Cytochemistry*, 25(7):741–753, 1977.