View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by UPCommons. Portal del coneixement obert de la UPC

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Optimización y paralelización del simulador del hardware de TaskSuperscalar

Trabajo Final de Grado

Autor: Cristian Morales Pérez

Director: Carlos Álvarez Martínez – DAC

Codirector: Daniel Jiménez González – DAC

Fecha de defensa: 5 de Diciembre 2014

Titulación: Grado en Ingeniería Informática

Especialidad: Ingeniería de Computadores

Centro: FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)

Universidad: UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) – BarcelonaTech

Quisiera dar las gracias a los dos directores por su paciencia y consejos en este proyecto y la motivación que influyen en sus respectivas asignaturas. Y sin olvidar dar las gracias a mi padre y madre por echar una mano en lo que mínimamente pueden en los peores momentos.

Abstract

The execution times of an optimized software and programmed with an awareness of architecture in which to be executed may vary considerably from those of a code with the same functionality but developed without considering these aspects. Therefore, the objective of this project is to apply the knowledge gained throughout the bachelor degree, both design and architecture of a computer and programming to reduce the execution time of a particular software. The design and functionality of the cache of a processor, strengths and weaknesses of a pipelined processor or the use of SIMD instructions represent diverse concepts which have been the clue in making various modifications to the source code.

One such modification has been parallelized code to use multiple processor cores simultaneously and thus add concurrency to the simulator. But in this case it has not been possible to parallelize the code because the design of the simulator causes that the overhead of using this type of programming is higher than the serial program execution.

The software which are improved his performance is the simulator of the *TaskSuperScalar* hardware. It is a hardware simulator currently under development, which will support the problem caused by the dependencies between tasks in parallel programming, trying to exploit to the maximum the parallelism between these tasks, synchronizing automatically depending on these dependencies. The executions of the original simulator can last hours depending on the input parameters. Therefore, a drastically reduction of these execution times will cause the support to the development of the *TaskSuperScalar* hardware by the simulator will be more effective.

Resumen

Los tiempos de ejecución de un software optimizado y programado teniendo consciencia de la arquitectura en la cual se va a ejecutar puede variar considerablemente respecto a los de un código con la misma funcionalidad pero desarrollado sin tener en cuenta estos aspectos. Por esta razón, el objetivo de este proyecto ha sido aplicar los conocimientos adquiridos durante toda la carrera, tanto del diseño de la arquitectura de un computador como de programación para reducir los tiempos de ejecución de un software en concreto. El diseño y la funcionalidad de la memoria caché de un procesador, ventajas y flaquezas de un procesador segmentado o el uso de instrucciones vectoriales representan diversos conocimientos que han sido clave a la hora de hacer las diversas modificaciones al código fuente.

Una de estas modificaciones ha sido paralelizar el código con el fin de utilizar múltiples núcleos de un procesador simultáneamente y así añadir concurrencia al simulador. Pero en este caso no ha sido posible paralelizar el código ya que por el diseño del simulador, el coste en tiempo de utilizar este tipo de programación es superior a la misma ejecución del programa.

El software que se ha optimizado para aumentar el rendimiento es el simulador del hardware *TaskSuperscalar*. Es un simulador de un hardware que está actualmente en desarrollo, el cual dará soporte al problema que suponen las dependencias entre tareas en la programación paralela intentando explotar al máximo el paralelismo existente entre tareas, sincronizándolas automáticamente en función de estas dependencias. Las ejecuciones de la versión original simulador pueden llegar a durar horas dependiendo de los parámetros de entrada que se le apliquen, por lo tanto reducir drásticamente este tiempo hará que el soporte que al desarrollo del *TaskSuperscalar* por parte del simulador sea más eficaz.

Resum

Els temps d'execució d'un software optimitzat i programat tenint consciència de l'arquitectura en la qual s'executarà pot variar considerablement respecte als d'un codi amb la mateixa funcionalitat però desenvolupat sense tenir en compte aquests aspectes. Per aquesta raó, l'objectiu d'aquest projecte ha estat aplicar els coneixements adquirits durant tota la carrera, tant del disseny de l'arquitectura d'un computador com de programació per reduir els temps d'execució d'un programari en concret. El disseny i la funcionalitat de la memòria caché d'un processador, avantatges i febleses d'un processador segmentat o l'ús d'instruccions vectorials representen diversos coneixements que han estat clau a l'hora de fer les diverses modificacions al codi font.

Una d'aquestes modificacions ha estat paral·lelitzar el codi per tal d'utilitzar múltiples nuclis d'un processador simultàniament i així afegir concurrència al simulador. Però en aquest cas no ha estat possible paral·lelitzar el codi ja que pel disseny del simulador, el cost en temps d'utilitzar aquest tipus de programació és superior a la mateixa execució del programa.

El software que s'ha optimitzat per augmentar el rendiment és el simulador del hardware *TaskSuperscalar*. És un simulador d'un hardware que està actualment en desenvolupament, el qual donarà suport al problema que suposen les dependències entre tasques en la programació paral·lela intentant explotar al màxim el paral·lelisme existent entre tasques, sincronintzan-les automàticament en funció d'aquestes dependències. Les execucions de la versió original del simulador poden arribar a durar hores depenent dels paràmetres d'entrada que se li apliquin, per tant reduir dràsticament aquest temps farà que el suport que dona el simulador al desenvolupament del *TaskSuperscalar* sigui més eficaç.

Índice de contenido

1. Introducción 13
1.1 Contexto 13
1.2 Estado del arte 14
1.3 Integración de conocimientos....................15
2. Planificación general 17
2.1 Objetivo y alcance 17
2.2 Planificación del proyecto....................17
2.3 Costes....................20
2.4 Leyes y regulaciones....................22
2.5 Sostenibilidad 23
3. Entorno de Trabajo 25
3.1 Metodología de experimentación 25
3.2 Características del PC 26
3.3 Compilador....................27
3.4 Profilers....................27
3.5 Ficheros de configuración....................28
3.6 Trazas....................28
3.7 Otras herramientas....................30
4. Técnicas aplicadas y resultados....................31
4.1 Fase de optimización 31
4.1.1 Replanteamiento del código – v1.1....................31
4.1.2 Optimización de memoria - v1.2 33
4.1.3 Bithacking – v1.3....................35
4.1.4 SIMD - v1.4....................36

4.1.5 Desenrolle de bucles e inlining de funciones - v1.5 37

4.1.6 Replanteamiento del código - v2.0 38

4.1.7 SIMD, Bithacks y desenrolle de bucles - v2.1 41

4.2 Paralelización 43

4.3 Resultados 47

5. Conclusiones 49

6. Trabajo futuro....50

7. Glosario....51

8. Abreviaturas 52

9. Bibliografía 53

Índice de tablas

Tabla 1: Duración de las fases 20
Tabla 2: Costes hardware iniciales 20
Tabla 3: Costes hardware finales 21
Tabla 4 : Gastos totales 22
Tabla 5: Información de las trazas utilizadas 29
Tabla 6: Tiempos de ejecución de la versión original 31
Tabla 7: Tiempos de ejecución de la v1.1 33
Tabla 8: SpeedUps de la v1.1 33
Tabla 9: Tiempos de ejecución de la v1.2 34
Tabla 10: SpeedUp de la v1.2 34
Tabla 11: Tiempos de ejecución de la v1.3 36
Tabla 12: SpeedUp de la v1.3 36
Tabla 13: Tiempos de ejecución de la v1.4 37
Tabla 14: SpeedUp de la v1.3 37
Tabla 15: Tiempos de ejecución de la v1.5 38
Tabla 16: SpeedUp de la v1.5 38
Tabla 17: Tiempos de ejecución de la v2.0 41
Tabla 18: SpeedUp de la v2.0 41
Tabla 19: Tiempos de ejecución de la v2.1 42
Tabla 20: SpeedUp de la v2.1 42
Tabla 21: Tiempos de ejecución de la primera versión paralelizada 44
Tabla 22: SpeedUps de la primera versión paralelizada 44
Tabla 23: Tiempos de ejecución de la tercera versión paralelizada 46
Tabla 24: SpeedUps de la tercera versión paralelizada 46

Índice de figuras

Figura 1: Gantt Planificación inicial 19

Figura 2: Gantt Planificación actual 19

Figura 3: Configuraciones del SimTSS 28

Figura 4: Pseudocódigo de la función Sched original 32

Figura 5: Pseudocódigo de la función Sched mejorada 32

Figura 6: Función Worker original en pseudocódigo 35

Figura 7: Función Worker con bithacks en pseudocódigo 35

Figura 8: Aproximación diseño del simulador en pseudocódigo 43

Figura 9: Pseudocódigo de prototipo de paralelización 45

Figura 10: Comparación de SpeedUps con diferentes trazas 47

Figura 11: Comparación de SpeedUp entre versiones 48

1. Introducción

1.1 Contexto

El *TaskSuperscalar* (*TSS*) es un hardware en desarrollo, el cual intenta explotar al máximo el paralelismo existente entre tareas, sincronizándolas automáticamente en función de las dependencias determinadas por el programador/a[1][2].

El hardware está en desarrollo y por ello, se está haciendo pruebas con un simulador programado en C, desarrollado en el grupo de investigación dirigido por Daniel Jiménez y Carlos Álvarez. El simulador (*SimTSS*) emula el funcionamiento del *TSS* y sus componentes ciclo a ciclo.

El *SimTSS* es un simulador basado en trazas. Las trazas incluyen los datos y los meta-datos de las tareas (número de identificación de tarea, ciclos de ejecución, número de dependencias, dirección de dependencias y el ciclo de inicio de cada tarea). Estas trazas se obtienen instrumentando el código fuente de una aplicación para obtener el número de ciclos del reloj de la *CPU* y así obtener el nombre de ciclos necesarios de ejecución, los ciclos de inicialización para cada tarea y el número total de ciclos necesarios para ejecutar todas las tareas de forma secuencial.

Cuando la ejecución de la traza se completa, es decir, el simulador finaliza la ejecución, el *SimTSS* crea un archivo donde muestra en que ciclo ha finalizado cada tarea, el número total de ciclos necesarios y algunas estadísticas del uso de algunos componentes. La ejecución del simulador puede durar desde pocos segundos a varias horas, en función de la longitud de las trazas, el tipo de aplicación de donde se han obtenido las trazas y la complejidad del *TSS* que se haya configurado. Esta configuración viene dada por un archivo donde se define el diseño y las propiedades del TSS y de sus componentes a simular. A mayor número de componentes y conexiones entre ellos, mayor será el tiempo de simulación. La conexión entre componentes está compuesta de varios árbitros y *FIFOs*.

El objetivo de este proyecto es reducir al máximo los tiempos de ejecución actuales mediante técnicas de optimización y paralelización en el código fuente del simulador. Según la configuración del *TSS* y las trazas el tiempo de ejecución del simulador puede ser muy elevado, como se ha comentado anteriormente. Esto repercute en el soporte que proporciona el simulador al desarrollo del *TaskSuperscalar*.

Para reducir los tiempos de ejecución se aplican al código fuente original ciertas técnicas de optimización que modifican el código, pero sin llegar a modificar la funcionalidad de este. Aparte de estas técnicas, se ha intentado paralelizar el código con el fin de utilizar múltiples núcleos de un procesador simultáneamente y así añadir concurrencia al simulador.

1.2 Estado del arte

El simulador es una herramienta de un trabajo privado actualmente en desarrollo y por lo tanto este problema no está resuelto ni existe ningún software similar conocido. Por la naturaleza del proyecto (optimizar y paralelizar), se puede decir que el mismo simulador sin mejorar es una solución existente, pero no cumple las expectativas que se esperan de este proyecto respecto a los tiempos de ejecución.

No hay ningún simulador software del *TSS* a parte de la versión original pero existe un prototipo *FPGA (Field Porgrammable Gate Array)* del *TaskSuperscalar*[3]. Una *FPGA* es un dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada mediante un lenguaje de descripción especializado como puede ser *VHDL*. Con una *FPGA* se puede reproducir desde una simple puerta lógica hasta complejos *SoC (System-On-a-Chip)*. Esté prototipo es la primera implementación hardware de un prototipo del *TaskSuperscalar*. El prototipo funciona aproximadamente a 150Mhz en una FPGA comercial y puede llegar a simular la ejecución de 1024 tareas de forma simultánea dependiendo de sus dependencias.

Existen otros proyectos de optimizaciones[4] e implementaciones en *FGPA*, pero en este caso no tienen que ver con ni con el *TaskSuperscalar* ni con el *SimTSS*.

1.3 Integración de conocimientos

En el proyecto se han aplicado conocimientos aprendidos durante todo el grado, desde el primer curso al último.

En primer lugar, las primeras asignaturas de programación del grado (Programación 1 y Programación 2) han sido las que han otorgado los conocimientos básicos de la programación en general y de C/C++ en particular para desarrollar este proyecto. Pero la mayor ayuda que han proporcionado estas asignaturas es obtener la capacidad de resolución de problemas algorítmicos de una forma elegante y eficaz.

Desde el primer curso del grado se han realizado asignaturas que han influenciado en el desarrollo del proyecto, en este caso, asignaturas relacionadas con el estudio del diseño de la arquitectura de computadores (Introducción a los Computadores, Estructura de Computadores, Arquitectura de Computadores, Arquitectura de Computadores 2, Multiprocesadores). Es cierto que las dos primeras asignaturas son muy básicas, pero han sido esenciales para poder realizar las tres siguientes. Gracias a Arquitectura de Computadores hemos aprendido el código maquina x86, pero principalmente cómo funciona la memoria de un procesador, y esto ha ayudado a tener los conocimientos necesarios para optimizar eficientemente el uso de memoria en el software del proyecto. Lo mismo ocurre con la asignatura de Multiprocesadores, pero en este caso con el uso de memoria compartida entre diferentes procesadores y sus protocolos de coherencia y consistencia. Finalmente Arquitectura de Computadores 2 ha sido clave para entender el funcionamiento y diseño de un procesador segmentado y conocer algunas de las flaquezas de este tipo de diseño. Una de estas flaquezas es cuando el procesador no es capaz de predecir correctamente la condición de saltos condicionales, por ello en el proyecto se ha intentado reducir este tipo de instrucciones que aumentan el tiempo de ejecución en algunos casos.

También se han integrado al proyecto los conocimientos obtenidos en la asignatura obligatoria de grado que cubre los aspectos fundamentales relacionados con la programación paralela. De esta asignatura se han utilizado la mayoría de sus contenidos, como la capacidad de análisis de aplicación paralelas, principios de programación paralela y el uso de *OpenMP*. A parte de los conocimientos adquiridos en esta asignatura, también se está utilizando las diferentes herramientas utilizadas en esta

asignatura, como son *Tareador*, *Paraver* o el *clúster Boada* alojado en el centro de procesamiento de datos del Departamento de Arquitectura de Computadores de la facultad.

Finalmente, la asignatura con nombre Programación Consciente de la Arquitectura ha proporcionado la mayor motivación para realizar este proyecto y los conocimientos esenciales para desarrollar el proyecto. La mayoría de temas tratados en esta asignatura han sido aplicados en la versión definitiva actual del proyecto. Uno de estos temas es herramientas de programación y optimización, es decir, análisis de rendimiento del programa (*profiling*). Además, los temas de reducción de operaciones de larga latencia, optimización del control del flujo reduciendo saltos críticos, aplicando *inlining*, desenrollando bucles, etc., y el tema de optimización consciente de la memoria. Por último, el tema que trata las extensiones vectoriales. En la asignatura la colección de instrucciones vectoriales *SSE2*, pero en el caso del proyecto ha sido necesario utilizar la colección *SSE4.2* por falta de instrucciones en la colección anterior.

2. Planificación general

2.1 Objetivo y alcance

El objetivo del trabajo es optimizar y paralelizar el código fuente del simulador del TSS para reducir los tiempos de ejecución de la versión original. Este objetivo se ha alcanzado, pero con un pequeño detalle. Los tiempos de ejecución se han reducido hasta lo deseado inicialmente pero no se ha conseguido realizar una paralelización del código que reduzca los tiempos de ejecución. Es cierto que la paralelización esperada era el uso de alguna extensión la cual permitiera convertir el código secuencial a código multi-hilo, pero se ha conseguido paralelizar de una forma diferente, con vectorización.

La razón por la que no se ha obtenido una versión para ejecutarse con diferentes núcleos de un procesador es porque el coste en tiempo (*overhead*) de utilizar este tipo de programación es demasiado elevado y provoca que el programa no consiga ningún tipo de mejora en su rendimiento.

El alcance del proyecto planificado inicialmente era reducir el tiempo de ejecución de la mayoría de ejecuciones a una décima parte del original, es decir, conseguir un *SpeedUp de 10x*. Con las modificaciones hechas en el código, esta mejora ha sido alcanzada y superada en la mayoría de casos, alcanzando incluso *SpeedUps de casi 30X*, por lo tanto se ha alcanzado este objetivo. En la planificación inicial se pensaba que la mayor mejora se obtendría paralelizando, ya que a simple vista se reconocían diferentes bucles costosos que parecían ser funciones del código potencialmente paralelizables. Finalmente no ha sido posible paralelizar, y en contra de lo que se pensaba al principio, la mayor y única mejora se ha logrado utilizando optimizaciones de código.

2.2 Planificación del proyecto

Primeramente, se calculó que el proyecto finalizaría en junio de 2014, pero se tuvo que extender el tiempo necesario para el proyecto hasta diciembre de 2014 manteniendo las horas totales de dedicación al proyecto. La razón por la que se tuvo que aplazar la defensa del proyecto es la incompatibilidad con la vida laboral del estudiante ya que al principio se hizo la planificación teniendo en cuenta que el estudiante no tendría obligaciones laborales.

Para comenzar el proyecto, se realizó la fase de planificación cursando la asignatura de Gestión de Proyectos, la cual fue esencial para el funcionamiento correcto de las siguientes fases que componen el proyecto. En esta fase también se incluye las primeras reuniones hechas con los directores del proyecto.

Con la fase anterior finalizada, se dio paso a la fase de inicio y análisis. Esta fase consistió en la obtención del código fuente del simulador mediante el controlador de versiones *Git* y un análisis del diseño y la funcionalidad del *TaskSuperScalar* y del simulador para una completa comprensión de como emula el simulador al hardware. Esta fase no se vio afectada por ningún contratiempo y se realizó según lo esperado en la fase de planificación.

La siguiente fase que se realizó fue la fase de *Profiling*. Esta fase consistió en analizar el código con la ayuda de *profilers*. Un *profiler* es una herramienta que permite hacer un análisis de rendimiento del código fuente, y en este caso averiguar las regiones de funciones o líneas que consumen más tiempo de ejecución. Esta fase ha servido para focalizar el trabajo en las partes más costosas del simulador.

Las dos fases posteriores a la fase de *profiling* son las fases de desarrollo, fase de optimización y fase de paralelización. El objetivo de la primera ha sido reducir el tiempo de ejecución del simulador mediante diferentes técnicas como pueden ser reducción de operaciones de larga latencia, memorización, optimización del uso de memoria, reducción de saltos condicionales, o uso de instrucciones vectoriales. Finalizada esta fase se tiene una versión del software secuencial, es decir, se ejecuta en un solo núcleo del procesador, y por ello, en la siguiente fase se ha intentado explotar el paralelismo y utilizar diferentes núcleos de un procesador para realizar una ejecución del simulador.

Las fases de *profiling* y las dos siguientes fases, fase de optimización y fase de paralelización, se planificaron para desarrollarlas de forma secuencial como se puede observar en la Figura 1, es decir, primero hacer *profiling*, después optimización y finalmente paralelización del código. Es cierto que para desarrollar las fases de optimización y paralelización primero se debe hacer la de *profiling* para centras todos los esfuerzos en las partes más problemáticas y costosas del programa, pero también será necesario seguir haciendo análisis con *profilers* cuando se están ejecutando estas fases, para ver

los cambios que se producen en estos análisis al reducir los tiempos de ejecución con las diferentes técnicas aplicadas y así ir focalizando el trabajo de forma eficiente.

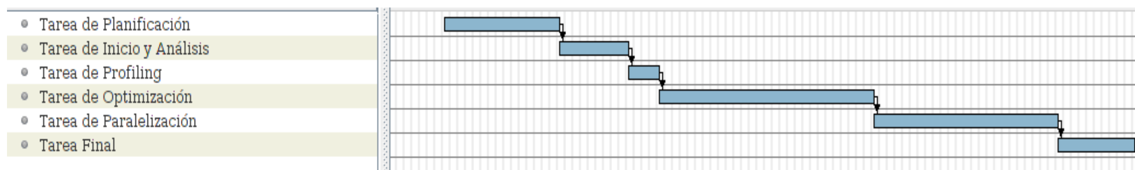

Figura 1: Gantt Planificación inicial

Principalmente hay dos cambios en la planificación, y éstos se pueden observar en el diagrama de Gantt hecho en la planificación inicial (Figura 1) y en el diagrama de Gantt que define la planificación final del proyecto (Figura 2). El primero de estos cambios es que la fase de *profiling* no finaliza hasta que la fase de paralelización y la fase de optimización no han finalizado, como se ha comentado anteriormente. El segundo cambio es que la fase de optimización empieza junto con la fase de *profiling*, justo cuando se finaliza la fase de análisis.

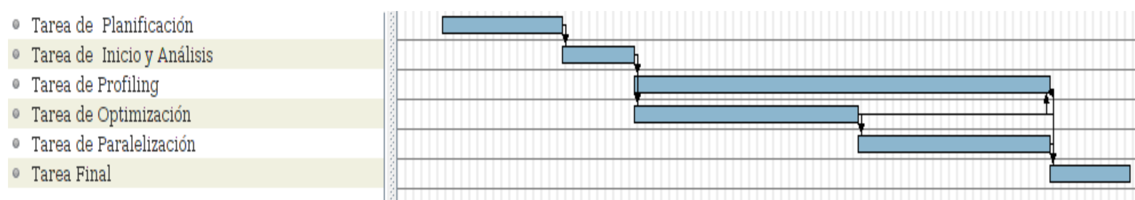

Figura 2: Gantt Planificación actual

Estos cambios no afectan al coste del proyecto ya que se siguen utilizando los mismos recursos, ni al tiempo necesario para cada fase. Los tiempos totales seguirán siendo iguales a los planificados solo que estarán distribuidos de forma diferente.

La fase final es en la que se está redactando esta memoria y se está haciendo la preparación de la defensa del proyecto. Esta fase concluye con la lectura del proyecto. Hasta el momento no ha surgido ningún problema.

Finalmente, como se puede observar en la Tabla 1, los tiempos no han sido modificados respecto a los estimados en la primera fase, y se ha necesitado aproximadamente el tiempo planificado.

Fase	Tiempo (horas)
Planificación	75
Inicio y Análisis	45
Profiling	20
Optimización	140
Paralelización	120
Final	50
TOTAL	**450**

Tabla 1: Duración de las fases

2.3 Costes

Los gastos han variado levemente respecto a los planificados al inicio del proyecto. Primero de todo, al haberse incrementado el tiempo de realización del proyecto de 4 meses a 9 meses, los costes relacionados con el hardware y su amortización han variado. Se desglosó inicialmente el coste de los diferentes componentes del ordenador como se puede observar en la Tabla 2.

Componente	Unidades	Precio/u (€)	Vida Útil	Precio amortizado (€)
AMD FX-6300	1	95,80	4 años	7,95
RAM Kingston 1600Mhz 4GB	2	33,30	4 años	5,55
Placa base Gigabyte	1	55,00	4 años	4,58
ATI Radeon HD 6770	1	101,05	4 años	8,42
Fuente Alimentacion 600W	1	40,12	4 años	3,34
HDD 1TB	1	42,65	4 años	3,55
Caja	1	37,45	4 años	3,12
Monitor LG 27 pulgadas	1	190,10	4 años	15,84
Teclado	1	75,00	4 años	6,25
Ratón	1	49,05	4 años	4,09
TOTAL				**62,74**

Tabla 2: Costes hardware iniciales

En la Tabla 2 se calcularon los precios amortizados suponiendo que el tiempo del proyecto no se extendería, y por lo tanto el uso de este ordenador sería de 4 meses. Finalmente la duración del proyecto ha sido de 9 meses y en la Tabla 3 se muestra el gasto final en hardware.

Componente	Unidades	Precio/u (€)	Vida Útil	Precio amortizado (€)
AMD FX-6300	1	95,80	4 años	17,96
RAM Kingston 1600Mhz 4GB	2	33,30	4 años	12,49
Placa base Gigabyte	1	55,00	4 años	10,31
ATI Radeon HD 6770	1	101,05	4 años	18,95
Fuente Alimentacion 600W	1	40,12	4 años	7,52
HDD 1TB	1	42,65	4 años	8,00
Caja	1	37,45	4 años	7,02
Monitor LG 27 pulgadas	1	190,10	4 años	35,64
Teclado	1	75,00	4 años	14,06
Ratón	1	49,05	4 años	9.2
TOTAL				**141,69**

Tabla 3: Costes hardware finales

En el caso de gastos en recursos software, se ha utilizado Software Libre en todo momento y en consecuencia se ha mantenido el gasto a cero. Los gastos estimados, ya que realmente no es un trabajo remunerado, destinados a recursos humanos tampoco han sufrido cambios. Inicialmente se estimó que serían necesarias 450 horas de trabajo por parte del estudiante y 50 horas por parte del director. Se estimó una remuneración ficticia de 15€/hora y 40€/h respectivamente.

Los gastos generales que engloban diferentes gastos como son la luz, internet, material fungible e impuestos varios, se calcula que se ha gastado una cantidad aproximada a la planificada inicialmente, unos 200€.

Finalmente, la suma total de los gastos aumenta de 9012,74€ a 9091,69€ (Tabla 4). Realmente el coste estimado en recursos humanos distorsiona el resultado dado que es un valor muy elevado comparado

con los demás gastos. Además, no existe ninguna remuneración económica de las horas cuantificadas. Eliminando este gasto, el coste final del proyecto sería de 341,69€ (262,74€ en la planificación inicial).

Presupuesto especifico	**Coste (€)**
Recursos Humanos	8750
Software	0
Hardware	141,69
Generales	200
TOTAL	**9091,69**

Tabla 4: Gastos totales

Se ha seguido el plan de control establecido en la fase inicial del proyecto para tener revisado en todo momento el gasto realizado y las horas empleadas y así tener una visión real del presupuesto disponible y poder reaccionar correctamente en caso de un improvisto, como podría ser el fallo de un componente hardware, la necesidad de actualización de alguno de estos componentes o la necesidad de algún software con licencia.

2.4 Leyes y regulaciones

El software del proyecto se encuentra bajo licencia GPL (*General Public License*), esto quiere decir que se garantiza a los usuarios finales la libertad de usar, estudiar, compartir y modificar el software. En este caso, no se distribuye públicamente puesto que pertenece al ámbito de la investigación. Usar la GPL exige que todas las versiones mejoradas que se publiquen sigan siendo software libre y esto evitará el riesgo de tener que competir con una versión modificada privativa del mismo trabajo. Por lo tanto, la versión del software desarrollada por el estudiante sigue siendo libre y no se ha convertido en un software privativo y da total libertad a que futuros usuarios utilicen esta versión para seguir mejorando el simulador.

La propiedad intelectual del trabajo está regulada por la normativa aprobada por el Consell de Govern (10/10/2008) por la cual se aprueba la confidencialidad, responsabilidad patrimonial y la propiedad industrial e intelectual a la Universitat Politècnica de Catalunya.

La titularidad del software desarrollado corresponde a la Universidad Politècnica de Catalunya, ya que se ha desarrollado en el marco de una actividad académica que ha estado dirigida y coordinada por profesorado de la UPC. También corresponde la titularidad de los derechos de explotación del software, y el estudiante y los profesores serán considerados coautores del mismo. En este caso, al ser un trabajo que se enfocada a la investigación no se plantea la explotación económica ni obtención de beneficio económico del proyecto. Por lo tanto, el director y codirector del proyecto, y todo su equipo de investigación, tendrán plena capacidad de utilizar mi versión del software para el uso que deseen de forma libre.

La versión original del simulador aún no está definida como Software Libre por el *Barcelona Supercomputing Center*, por lo tanto, no se adjuntará la versión optimizada del código como material adicional a esta memoria.

2.5 Sostenibilidad

Un impacto directo de la realización de este proyecto es la reducción de gasto energético a la hora de utilizar el simulador optimizado en futuras ejecuciones del simulador, dado que se han reducido los tiempos de ejecución de forma considerable como se ha comentado anteriormente. Por esta razón, un método para cuantificar los efectos medioambientales que repercute este proyecto es a partir de la mejora de redimiendo (*SpeedUp*) alcanzada y una estimación del tiempo de uso de este simulador en el futuro.

Pero hay que tener en cuenta, que para realizar este proyecto ha sido necesario en todo momento un ordenador, de manera que en total suman 450 horas de uso del PC para el proyecto. Si de media se ha obtenido una mejora de un 10X, es decir, se han reducido los tiempos de ejecución a una décima parte se debería tener planificado utilizar el simulador sin optimizar unas 500 horas. Así ese trabajo se realizaría en 50 horas (diez veces más rápido) y así ahorrarse 450 horas de ejecuciones y de uso de una computadora. Por lo tanto, para que el proyecto sea rentable medio-ambientalmente hablando se debería utilizar la versión optimizada del simulador más de 50 horas, según la mejora que se ha

obtenido (una media de 10X), y así compensar el gasto energético de la realización del proyecto. Este cálculo se basa en que las potencias son iguales del computador en el cual se ha desarrollado el proyecto y en el que se harán las ejecuciones del simulador optimizado.

Al reducir el gasto energético en las ejecuciones futuras del simulador, también existe un impacto económico directo y se verá reducido el gasto económico en energía en los proyectos donde se utilice el simulador. Cuantificar este ahorro económico se hace complicado ya que depende del precio de la energía donde se desarrolla el proyecto y donde se harán las ejecuciones del simulador mejorado y las diferentes potencias de las computadoras. Además, se debería tener en cuenta el gasto realizado en el desarrollo de este trabajo.

Respecto al impacto social, el uso de software libre en todo el proyecto se debe a su valor social fundamental, puesto que la única restricción que tiene es la de conservarse libre, lo cual quiere decir que puede ser explotado, verificado, reproducido, y extendido, en todas sus capacidades. El propio software que se desarrolla está bajo licencia GPL y un aspecto crucial bajo este tipo de licencia es que los usuarios tienen la libertad de cooperar. Es absolutamente esencial que a los usuarios que deseen ayudarse entre sí se les permita compartir sus correcciones de errores y mejoras con otros usuarios. Este proyecto refleja completamente lo que representa este tipo de licencia, se está cooperando para mejorar un software desarrollado por otras personas y finalmente compartiendo la versión mejorada con estas personas para que puedan aprovecharse de ella, y posiblemente, para que sigan aplicando mejoras.

3. Entorno de Trabajo

3.1 Metodología de experimentación

Para poder conseguir la mejora deseada en los tiempos de ejecución, se han seguido los siguientes métodos. Primero de todo, ha sido esencial la comprensión de la estructura del TaskSuperscalar y sus diferentes componentes para entender como es simulado por el SimTSS y que métodos de entrada y salida tiene.

Cuando ya se tiene una idea general de cómo funciona el simulador, se ha pasado a analizar el código con diferentes herramientas de *profiling* (el proyecto tiene una fase dedicada expresamente a este tipo de análisis por su importancia). Mediante estas herramientas se calcula el tiempo de ejecución dedicado a las funciones o líneas y así descubrir cuáles son los puntos más conflictivos del programa. También se capturan diferentes eventos del programa como fallos en la predicción de un salto o fallos en memoria cache, etc.

Teniendo ya definido donde se consume más tiempo de ejecución, se pasa a descubrir que está provocando esta situación (problema con memoria, instrucción de larga latencia, código redundante, etc.) e intentar solventarla aplicando técnicas de optimización y paralelización.

Cuando se obtiene una versión modificada del simulador, se comprueba cómo cambian los tiempos de ejecución para verificar si mejora o empeora el rendimiento. En caso de empeorar, se vuelve a analizar el código con *profilers* para intentar descubrir que ocurre y así poder evolucionar la modificación para acabar obteniendo una mejora de rendimiento o acabar descartando la modificación realizada. Para dar por buena una modificación se debe validar que los cambios que se realizan en el simulador no alteran el funcionamiento del simulador comprobando el canal de salida del programa, pero en particular comprobando el archivo donde el programa escribe en orden las tareas que van finalizando con el ciclo en el cual han finalizado y cuantos ciclos han necesitado para ejecutarse. No se comprueba que el archivo del original sea idéntico que el de la de la versión modificada, sino que se comprueba que la mayoría de tareas se hayan ejecutado en el mismo orden o con una pequeña variación de posición y que los ciclos de finalización sean los mismos o con algunos ciclos de más o de menos

respecto a los de la versión original.

Los tiempos de las ejecuciones se han obtenido mediante el comando /usr/bin/time (y en algunos casos el comando time) que ofrece Ubuntu. Este comando nos proporciona el tiempo de ejecución de usuario, de sistema y tiempo total con una la posibilidad de personalizar la precisión. También ofrece el tanto por ciento de dedicación de la CPU al proceso en concreto.

La metodología utilizada para obtener todos los resultados es hacer la media aritmética de los tiempos de 5 ejecuciones escogidas entre los tiempos de 7 ejecuciones donde se han eliminado las que tienen el tiempo máximo y mínimo. Si una de estas 7 ejecuciones tiene menos de un 99% de dedicación de CPU al proceso del simulador es descartada. Se ha cuidado no estar haciendo otras cosas en el ordenador mientras se hacen ejecuciones que pueda afectar e incrementen levemente los tiempos de ejecución. En el caso de las ejecuciones para profiling no solo se ha realizado una ejecución, ya que son mucho más costosas estas ejecuciones y no es necesaria una precisión perfecta.

3.2 Características del PC

Las características de la maquina “*domestica*” que se ha utilizado para hacer las ejecuciones y obtener los resultados de las diferentes optimizaciones es:

- Procesador: AMD FX-6300
  - 6 cores (1 thread por core)
  - SIMD: AVX & SSE4.2
- Memoria: 8GB de RAM
- Sistema Operativo: Ubuntu 14.04 64 bits

También se ha utilizado el clúster Boada, alojado en el Centro de Procesamiento de Datos del Departamento de Arquitectura de Computadores para utilizar herramientas de análisis alojadas en este servidor. Las características de Boada son:

- 2 procesadores Intel Xeon E5645 a 2.5 GHz
  - 6 Cores por procesador
  - 12 Threads por procesador
- 24 GB de RAM

3.3 Compilador

La versión de *gcc* que se ha utilizado es la *4.9.2*. Respecto a las opciones de compilación, inicialmente se empezó utilizando la opción de compilación *-O2* en contra de *-O3*, ya que aplica optimizaciones que mejoran considerablemente los tiempos de ejecución y no aplica optimizaciones al código aumentando considerablemente el tamaño del binario (desenrollando bucles, haciendo *inlining* para muchas funciones, etc.) que realizaría la opción de compilación *-O3*.

Finalmente, después de hacer diferentes pruebas, se decidió utilizar la opción de compilación *-O3*, ya que no se encontró ningún problema en las mejoras hechas por esta opción de compilación y así ahorrarnos de implementar las mejoras que ya hace está opción.

Se ha utilizado la opción de compilación -msse4.2 para poder utilizar las instrucciones *SIMD*. También se ha tenido que utilizar la opción *-fopenmp* para el uso de *OpenMP*. Se utiliza la versión 4.9.2 de *gcc* para poder utilizar *OpenMP* 4.0 e intentar aprovechar sus nuevas características. Además se ha hecho uso de la opción –S para obtener el código ensamblador que genera el compilador y así poder saber que instrucciones que se están ejecutando o que mejoras ya realiza el propio compilador.

3.4 Profilers

Los *profilers* son herramientas que permiten analizar el código mediante recopilación de eventos del programa como interrupciones por hardware o instrumentación del código. Gracias a esto se obtienen datos sobre que funciones o líneas de código consumen más tiempo de ejecución, cálculos sobre predicciones fallidas por parte del procesador, fallos de caché u otros eventos.

Inicialmente estaba planificado utilizar *Oprofile* y *Gprof* para realizar esta fase, pero utilizando *grpof* para analizar las funciones del código que consumen más tiempo de ejecución se comprobó que los resultados que daba *gprof* no eran correctos. El error es que muestra como una de las funciones que consume más tiempo de ejecución una función que no se llama en ningún momento en el simulador. Por lo tanto se decidió utilizar *QCachegrind*, y se han obtenido resultados similares pero sin este error anterior. También se han utilizado funciones de Linux para hacer *timing* de diferentes partes del código y comprobar que los resultados eran los esperados.

Este cambio no afecta al coste del proyecto ya que se sigue trabajando con OpenSource y tampoco incrementa el tiempo empleado en esta fase.

Las versiones de los tres profilers son:

- gprof 2.24
- OProfile 1.0
- QCacheGrind 0.7.4

3.5 Ficheros de configuración

El fichero de configuración, como se ha comentado anteriormente, define el diseño y las propiedades del TaskSuperscalar y de sus componentes. A mayor número de componentes y conexiones entre ellos, mayor será el tiempo de simulación. Las pruebas se han hecho con tres archivos de configuración diferentes, los cuales se pueden observar en la Figura 3.

#Config 1	#Config 4	#Config 8
WORKERs=16	WORKERs= 256	WORKERs= 1024
TRSs= 1	TRSs= 4	TRSs= 8
TRSCAPACITYTASKs= 256	TRSCAPACITYTASKs= 256	TRSCAPACITYTASKs= 512
eORTs= 1	eORTs= 4	eORTs= 8
OVTMEMLENGTH= 512	OVTMEMLENGTH= 512	OVTMEMLENGTH= 1024
ORTMEMLENGTH= 64	ORTMEMLENGTH= 64	ORTMEMLENGTH= 64
ORTMEMWAYs= 8	ORTMEMWAYs= 8	ORTMEMWAYs= 16
ORTMEMACCESSMODE= 3	ORTMEMACCESSMODE= 3	ORTMEMACCESSMODE= 3

Figura 3: Configuraciones del SimTSS

Los campos del archivo de configuración *WORKERs*, *TRSs* y *eORTs* determinan la cantidad de componentes de cada tipo que se emularan. Los demás campos determinan diferentes características de los componentes.

3.6 Trazas

En el fichero de configuración del simulador también está definido el archivo de entrada con las trazas. Las trazas incluyen los datos y meta-datos de las tareas (identificación, ciclos de ejecución, numero de dependencias, dirección de las dependencias y tiempo de inicio de cada tarea) obtenidas instrumentando el código de las aplicaciones.

Para realizar las ejecuciones con los diferentes ficheros de configuración se han utilizado diferentes

trazas de diferentes tamaños y aplicaciones:

Factorización Cholesky. La factorización de *Cholesky* descompone una matriz simétrica definida positiva A en el producto de L por L traspuesta, donde L es una matriz triangular inferior (A = LL'). La implementación divide A en b x b bloques o m x b paneles. Un bloque o panel es dividido en *subpaneles* que contienen t columnas. Se utiliza una versión de descomposición "*left-looking*". Las nombre trazas de esta aplicación empieza por "*llchol-trace-*".

LU. La factorización LU descompone una matriz m x n (m debe ser mayor o igual que n) en A = L * U, donde L es una matriz triangular inferior (m x n) y U es una matriz triangular superior (n x n). Es utilizada normalmente en sistemas de equitaciones lineales. Las nombre trazas de esta aplicación empieza por "*lu-trace-".*

Sparse LU Descomposition. Como la anterior, esta aplicación realiza una descomposición L, pero ahora sobra una matriz dispersa cuadrada. La matriz está almacenada en bloques de memoria contigua. Las nombre trazas de esta aplicación empieza por "*sparselu-trace-".*

Heat Diffusion. Esta es una implementación de método iterativo para la distribución de calor. Hay tres algoritmos para el usuario, *Jacobi*, *Gauss-Seidel* y *Red-Black*. En este caso se ha utilizado *Gauss-Seidel*. Las nombre trazas de esta aplicación empieza por "*heat-trace-*".

En la Tabla 5 se muestran el tamaño del archivo de las trazas utilizadas, los ciclos que serían necesarios para hacer una ejecución en serie de todas las tareas, los ciclos necesarios con las diferentes configuraciones y el número total de tareas.

	Tamaño (MB)	Ciclos totales en serie	Ciclos Config1	Ciclos Config4	Ciclos Config8	Numero de tareas
llchol-trace-2048-8	209.5	6493126936	2020372329	638220982	138582353	2829056
llchol-trace-2048-32	3.3	2138224964	188454268	48886095	12497972	45760
llchol-trace-2048-64	0.4	2682876680	174939161	37462436	31202235	5984
llchol-trace-2048-128	0.53	2164331212	158292086	86994678	86994674	816
sparselu-trace-2048-8	40.0	1031525632	71898626	27484061	26655545	707392
heat-trace-2048-256	6.9	352132440	282572182	115358927	22869209	65536
lu-trace-2048-4	7.4	19904014508	1271811809	184817189	147116144	131327

Tabla 5: Información de las trazas utilizadas

El primer número de las trazas determina el tamaño de la matriz, 2048x2048 en todos los casos. El segundo número determina el tamaño del bloque de la descomposición. Por lo tanto, cuanto más pequeño sea el bloque se descomposición, más tareas serán necesarias.

3.7 Otras herramientas

Tareador, alojado en el clúster Boada, es una herramienta que mediante la instrumentación permite definir regiones de código como tareas y que *Tareador* analice las dependencias que existen entre las tareas definidas por el programador/a.

Se han encontrado problemas a la hora de analizar el proyecto con *Tareador* por el gran tamaño del proyecto y el diseño que tiene el código. A raíz de estos problemas se ha intentado adaptar el análisis para obtener resultados útiles para el desarrollo del proyecto.

Se han implementado dos scripts para obtener los tiempos de ejecución y *profilings* para cada traza, con las tres configuraciones diferentes. En el caso de los tiempos de ejecución se hace 7 ejecuciones para cada caso como se ha dicho anteriormente, y así obtener un resultado fiable.

4. Técnicas aplicadas y resultados

4.1 Fase de optimización

En esta fase del proyecto, se han utilizado diferentes técnicas que modifican el código fuente original con tal de mejorar el rendimiento del simulador. Hay algunos cambios que no han servido para reducir los tiempos de ejecución pero han servido para evolucionar hacia un nuevo cambio o han sido descartados directamente.

Si no se dice lo contrario, los resultados mostrados son de ejecuciones del simulador con diferentes configuraciones y con trazas de diferente tamaño de la aplicación *Cholesky*, en concreto son las trazas "llchol-trace-2048-32", "llchol-trace-2048-64" y "llchol-trace-2048-128". Se han escogido estas trazas porque son medianamente representativas y no son demasiado largas para hacer varias ejecuciones seguidas. También se ha hecho una copia de la versión original para mantenerla y poder comparar los tiempos de ejecución y los resultados de la simulación. En la Tabla 6 se muestran los tiempos de ejecución de la versión original ejecutados con la opción de compilación *–O3* con las tres configuraciones diferentes y trazas de diferente tamaño de la aplicación *Cholesky*.

	Tiempo llchol-trace-2048-32 (s)	Tiempo llchol-trace-2048-64 (s)	Tiempo llchol-trace-2048-128 (s)
Config - 1	28,3	22,45	19,11
Config - 4	70,95	54,2	126,21
Config - 8	75,14	172,45	495,6

Tabla 6: Tiempos de ejecución de la versión original

4.1.1 Replanteamiento del código – v1.1

Antes de comenzar a optimizar, se ha analizado el código fuente con herramientas de profiling y los resultados muestran que entre un 40% y un 65% (según el tipo de traza y la configuración del TSS) del tiempo de ejecución lo consume la función llamada "*Sched*". En el código original, esta función está compuesta de dos búsquedas en dos bucles diferentes sobre el mismo vector. La segunda búsqueda solo se ejecuta si la primera búsqueda falla y no encuentra lo que busca. En la Figura 4 esta explicada su funcionalidad con pseudocódigo.

```
for(i = 0; i < N;++i)
    if (busqueda1)
        foo(i)
        return()
for(i = 0; i < N;++i)
    if (busqueda2)
        foo2(i)
        return()
```

Figura 4: Pseudocódigo de la función sched original

El cambio que se decide aplicar a esta función es hacer las dos búsquedas sin tener que pasar dos veces por las mismas posiciones del vector. Para hacer esto, en el primer bucle se comprueba si se cumple de la primera búsqueda y la segunda simultáneamente, pero si se cumple la condición de la segunda búsqueda solo se guarda la posición del vector donde se ha encontrado el caso de la `busqueda2` y se finaliza este bucle. Al salir del primer bucle, se sigue en el segundo bucle haciendo la primera búsqueda desde la posición en la que se quedó el anterior bucle. Si el segundo bucle finaliza sin encontrar la primera búsqueda se comprueba si se ha guardado alguna posición en la cual la segunda basquead se cumple. Si es así, se ejecuta el código correspondiente a la segunda búsqueda. En la Figura 5 está explicada su funcionalidad con pseudocódigo.

```
position = -1
for(i = 0; i < N;++i)
    if (busqueda1)
        foo(i)
        return()
    if (busqueda2)
        position = i
        break
for(; i < N;++i)
    if (busqueda1)
        foo(i)
        return()
if (position >= 0) foo2(position)
```

Figura 5: Pseudocódigo de la función sched mejorada

Con esta optimzación se obtiene una pequeña mejora en los tiempos de ejecución debido a la reducción de iteraciones y accesos a memoria en algunos casos. En las siguientes tablas se puede ver los tiempos obtenidos (Tabla 7) y el SpeedUp respecto a la versión original (Tabla 8) obtenido:

	Tiempo llchol-trace-2048-32 (s)	Tiempo llchol-trace-2048-64 (s)	Tiempo llchol-trace-2048-128 (s)
Config - 1	24,7	20,29	16,51
Config - 4	50,4	38,5	83,67
Config - 8	52,12	118,78	338,88

Tabla 7: Tiempos de ejecución de la v1.1

	SpeedUp llchol-trace-2048-32	SpeedUp llchol-trace-2048-64	SpeedUp llchol-trace-2048-128
Config - 1	1,15	1,11	1,16
Config - 4	1,41	1,41	1,51
Config - 8	1,44	1,45	1,46

Tabla 8: SpeedUps de la v1.1

4.1.2 Optimización de memoria - v1.2

Con la primera técnica ya aplicada, las dos funciones que consumen más tiempo de ejecución, entre un 70 y un 90%, recorren la misma estructura de datos. Una de estas funciones es la de la versión anterior (función Sched), y la segunda es un recorrido sobre todo el vector y haciendo escrituras si se cumple cierta condición (función Worker).

El vector que recorren las dos funciones más costosas es un vector de *structs*. Este *scruct* está definido de tal forma que ocupa 40 bytes y es posible reducirlo a 32 bytes reorganizando las variables. Al ver que la mejora de tiempo es mínima con esta mejora, al estar tratando de optimizar el uso de memoria y en concreto de este struct se decide cambiar la estrategia a la hora de definirlo. Este *struct* representa los procesadores que ejecutan las tareas definidas en las trazas y contiene diferentes variables (estado ocupado del procesador, identificación de posible tarea corriendo sobre este

procesador, estado finalizado de tarea, tiempo de ejecución necesario en ciclos de la tarea y ciclo de inicio de la tarea).

El cambio que se realiza es pasar de tener un vector de una estructura con cinco variables a tener cinco vectores de las diferentes variables del *struct*. Con este cambio se pretende explotar la localidad espacial, es decir, si una localización de memoria es referenciada en un momento concreto, es probable que las localizaciones cercanas a ella sean también referenciadas pronto. Esta característica antes no se daba. En los bucles de las dos funciones, la mayor parte del tiempo se utilizaba en hacer una lectura de la variable de estado ocupado de todos los procesadores. En el código, esta variable estaba distanciada de la siguiente a 40 bytes. En cambio ahora están una detrás de la otra y se traen a caché en bloque y así se producen menos fallos de caché.

En la tabla 9 se pueden observar los tiempos de ejecución con la optimización de memoria realizada mejora levemente y en la Tabla 10 está el SpeedUp alcanzado respecto a la versión original.

	Tiempo llchol-trace-2048-32 (s)	Tiempo llchol-trace-2048-64 (s)	Tiempo llchol-trace-2048-128 (s)
Config - 1	24,12	19,29	15,51
Config - 4	38,4	27,5	58,67
Config - 8	39,12	86,78	242,88

Tabla 9: Tiempos de ejecución de la v1.2

	SpeedUp llchol-trace-2048-32	SpeedUp llchol-trace-2048-64	SpeedUp llchol-trace-2048-128
Config - 1	1,17	1,16	1,23
Config - 4	1,85	1,97	2,15
Config - 8	1,92	1,99	2,04

Tabla 10: SpeedUp de la v1.2

Este cambio sigue sin mejorar demasiado, pero gracias a esta modificación se da la posibilidad de aplicar las siguientes técnicas.

4.1.3 Bithacking – v1.3

En el bucle de la función *Worker*, comentada anteriormente, en cada iteración hay un salto condicional al hacer la comprobación de una condición. El procesador predice la condición y sigue con la ejecución y si esta predicción es incorrecta, el procesador tiene que deshacer el trabajo hecho y ejecutar las instrucciones que debería haber ejecutado. Por lo tanto, se pierde bastante tiempo si el procesador no es capaz de predecir la mayoría de saltos. La decisión que se ha tomado ha sido aplicar la técnica de *bithacking*. Con esta técnica se pretende eliminar el salto condicional utilizando instrucciones aritmético-lógicas para manipular bits y finalmente conseguir el mismo resultado.

Originalmente, la función en cuestión (Figura 6) comprueba en la estructura de datos que simulan los procesadores, si está ocupado el procesador con una tarea y si esta tarea se ha estado ejecutando los ciclos necesarios. Si esto se cumple el procesador deja de estar ocupado y la tarea pasa a estar finalizada.

```
if (procesador_ocupado)
      if (ciclo_actual - ciclo_inicio_ >= ciclos_necesarios)
           procesador_ocupado = 0
           tarea_finalizada = 1
```

Figura 6: Función Worker original en pseudocódigo

Con la modificación aplicada, primero de todo se hace una comparación para saber si el ciclo actual del simulador menos el ciclo en el que ha empezado la tarea asignada a este procesador es mayor que los ciclos necesarios. Está comparación devuelve uno o cero, según si se cumple o no se cumple la comparación realizada. Como las variables de procesador ocupado y tarea finalizada solo pueden valer cero o uno, como el resultado de la comparación, mediante operaciones lógicas se consigue el mismo resultado. En la Figura 7 se puede observar cómo se hacen estas operaciones con pseudocódigo.

```
fin = ciclo_actual - ciclo_inicio >= ciclos_necesarios
no_fin = (fin + 1) AND 1  // negación del primer bit de fin
tarea_finalizada = tarea_finalizada OR ( fin AND procesador_ocupado)
procesador_ocupado =    procesador_ocupado AND no_fin
```

Figura 7: Función Worker con bithacks en pseudocódigo

Como se puede observar en la Tabla 11 los tiempos han empeorado y por tanto los SpeedUp respecto a la versión original (Tabla 12) se han visto afectados. Esto es debido a que en la versión original se hacen escrituras en memoria solo si se cumplen las dos condiciones, en cambio en la versión con bithacking en todas las iteraciones se hacen escrituras en memoria, y por ello el rendimiento del programa se ha visto reducido.

	Tiempo llchol-trace-2048-32 (s)	Tiempo llchol-trace-2048-64 (s)	Tiempo llchol-trace-2048-128 (s)
Config - 1	26,8	22,9	19,59
Config - 4	60,9	45,56	102,3
Config - 8	60,7	182,7	446,4

Tabla 11: Tiempos de ejecución de la v1.3

	SpeedUp llchol-trace-2048-32	SpeedUp llchol-trace-2048-64	SpeedUp llchol-trace-2048-128
Config - 1	1,06	0,98	0,98
Config - 4	1,17	1,19	1,23
Config - 8	1,24	0,94	1,11

Tabla 12: SpeedUp de la v1.3

4.1.4 SIMD - v1.4

Al no conseguir la mejora de tiempo deseada con la técnica de bithacking, se decide utilizar vectorización, para trabajar con instrucciones SIMD y registros de 128 bits. La vectorización es un tipo de paralelización del código que utiliza instrucciones vectoriales que trabajan sobre registros de 128, 256 o 512 bits. Estas instrucciones permiten a un solo núcleo del procesador trabajar con varios operandos de forma simultánea.

En este caso, el procesador en el que se ejecutará el simulador dispone de las colecciones de instrucciones SSE4.2 y AVX. La colección SSE4.2 habilita operaciones con registros de 128 bits, los cuales permiten trabajar, por ejemplo, con cuatro operandos de 32 bits o dos de 64 bits de forma simultánea. En cambio AVX habilita operar con registros de 256 bits, lo cual duplica la cantidad de operandos que se pueden utilizar de forma simultánea. El problema de AVX es que no dispone de

instrucciones de comparación necesaria para el caso del SimTSS, las cuales están disponibles en la colección de instrucciones vectoriales avanzadas AVX2, disponible en las arquitecturas Haswell de Intel y posteriores. En la Tabla 13 se puede observar reducción de tiempo respecto a la versión con Bithacks y en la Tabla 14 el SpeedUp respecto a la versión original.

	Tiempo llchol-trace-2048-32 (s)	Tiempo llchol-trace-2048-64 (s)	Tiempo llchol-trace-2048-128 (s)
Config - 1	20,96	16,73	13,6
Config - 4	28,1	21,13	45,02
Config - 8	24,88	56,81	148

Tabla 13: Tiempos de ejecución de la v1.4

	SpeedUp llchol-trace-2048-32	SpeedUp llchol-trace-2048-64	SpeedUp llchol-trace-2048-128
Config - 1	1,35	1,34	1,41
Config - 4	2,52	2,57	2,8
Config - 8	3,02	3,04	3,35

Tabla 14: SpeedUp de la v1.4

4.1.5 Desenrolle de bucles e inlining *de funciones - v1.5*

Con esta última mejora se consigue una mejora de rendimiento y por lo tanto se decide aplicar esta mejora a la versión anterior y seguir mejorarando por otras vías esta función. Una de estas estrategias es utilizar el desenrolle de bucles en el bucle en cuestión (en la función Worker).

El desenrolle de bucles consiste en hacer que el número de veces que se ejecuta el cuerpo del bucle sea menor, modificando este de tal margen que se ejecuten las mismas operaciones, teniendo especial cuidado con los índices que se manejan. Al aplicar esta técnica se han hecho pruebas con diferentes niveles de desenrolle para comprobar cuál es el óptimo.

Con el desenrolle del bucle, ha mejorado levemente el rendimiento gracias al a reducción de *overhead* en el control del bucle con. Otro *overhead* que se puede reducir, es el de la llamada a esta misma función utilizando la técnica de *inlining* y así no tener que sufrir el coste del paso de parámetros y del salto incondicional a la función. En la tabla 15 se puede comprobar que los *SpeedUp* no cambian de una forma considerable de la versión con SIMD, pero la mayoría de tiempos (Tabla 16) están por debajo de los obtenidos en la versión con instrucciones vectoriales.

	Tiempo llchol-trace-2048-32 (s)	Tiempo llchol-trace-2048-64 (s)	Tiempo llchol-trace-2048-128 (s)
Config - 1	20,45	16,44	13,2
Config - 4	27,21	20,7	43,74
Config - 8	24,18	55,31	147,2

Tabla 15: Tiempos de ejecución de la v1.5

	SpeedUp llchol-trace-2048-32	SpeedUp llchol-trace-2048-64	SpeedUp llchol-trace-2048-128
Config - 1	1,38	1,37	1,45
Config - 4	2,61	2,62	2,89
Config - 8	3,11	3,12	3,37

Tabla 16: SpeedUp de la v1.5

4.1.6 Replanteamiento del código - v2.0

Hasta el momento siempre se han realizado mejoras sobre las funciones que tratan la estructura de datos que representa los procesadores, ya que son las funciones que consumen más tiempo de ejecución. Esto implica que las mejoras de rendimiento que se obtienen con estas modificaciones dependen del tamaño del vector que representa a los procesadores. En los diferentes resultados de las versiones anteriores se puede observar cómo según aumenta los procesadores que tiene cada configuración, aumenta el *SpeedUp* obtenido.

Al ver que la estructura de datos que simula los procesadores es la que causa mayor conflicto y las dos únicas funciones que tratan con esta estructura son las dos que consumen más tiempo de ejecución

total se sugiere replantear totalmente esta estructura y las funciones que trabajan con ella. Una de estas funciones es la que se ha comentado en la primera mejora hecha al simulador. Esta función, primero busca si alguna tarea en ejecución en un procesador tiene la variable de estado de finalización activada para expulsarla del procesador y dejarlo libre. Si no encuentra ninguna tarea que cumpla esta condición, busca si hay algún procesador libre para intentar asignarle una tarea nueva. Para hacer estas dos búsquedas, se debía recorrer el vector hasta encontrar algún procesador que cumpla la condición en cada caso. La otra función que trata con esta estructura de datos es la que comprueba en todos los procesadores con tareas asignadas si estas tareas han finalizado o no y así activar la variable de estado finalizado de la tarea.

Como se ha dicho anteriormente, dos de los campos de la estructura de datos que representaban a los procesadores (`Workers`) eran la variable de estado que determinaba si estaba ocupado el procesador y la variable de estado que determinaba si la tarea asignada al procesador estaba finalizada. Estos dos valores solo pueden valer uno o cero para determinar ocupado o no ocupado y finalizada o no finalizada respectivamente. Que el procesador esté ocupado quiere decir que hay una tarea ejecutándose en él. Que una tarea asignada en un procesador esté finalizada quiere decir que el número de ciclo actual es mayor que la suma del ciclo de inicio de la tarea y el número de ciclos necesarios para ejecutar la tarea.

Desde que se le asigna a un procesador una tarea hasta que este procesador pasa a estar libre, el tiempo de inicio de tarea y el tiempo necesario de ejecución no cambian, y lo que realmente nos importa es saber es si esta tarea ha finalizado o no en el ciclo actual. Está comprobación se hace en la función más costosa de todo el programa, como se ha comentado anteriormente (función *`Worker`*), pero a la misma vez es la más simple. En esta función, por cada procesador que está ocupado (tiene una tarea asignada), se comprueba que el número de ciclo actual es mayor que la suma del ciclo de inicio de la tarea y el número de ciclos necesarios para ejecutar la tarea, y si es así el la variable que determina si una tarea ha finalizado pasa a ser uno. Esto quiere decir que lo único que nos importa es saber cuándo finaliza la tarea, su identificador, su estado y el estado del procesador.

El primer cambio que se plantea es eliminar del *`struct`* que define el procesador las variables de estado de tarea finalizada, de ciclo de inicio y ciclos necesarios y añadir una variable que represente

los ciclos restantes para finalizar la tarea. Con este cambio se sabrá si una tarea está finalizada si el tiempo restante es cero y así el simulador no tiene que hacer la comprobación cada ciclo para cada procesador con una tarea asignada de que el número de ciclo actual es mayor que la suma del ciclo de inicio de la tarea y el número de ciclos necesarios para ejecutar la tarea, simplemente en cada ciclo se realiza una resta saturada (el valor mínimo del resultado de la resta es cero) en una unidad los ciclos restantes para finalizar.

Después de este cambio, también se plantea eliminar la variable que determina si un procesador está ocupado y en su lugar se decide tener un vector ordenado con una variable global que determina cuantos procesadores están ocupados con una tarea asignada. Por lo tanto, los primeros procesadores del vector serán los que están trabajando, hasta la posición que marque a la variable creada. El orden del vector lo determina la variable de ciclos restantes de la tarea de cada procesador.

Se decide ordenarlo de forma descendente para que las tareas que se tienen que ir sacando sean las últimas, y solo se tendrá que reducir la variable que define cuantas tareas están ejecutando cuando se expulse una tarea. Con este cambio se ha pasado de tener un vector que representan procesadores a tener una lista ordenada de tareas en ejecución. Pero hay que tener en cuenta no tener más tareas en ejecución que procesadores emulados tenga la configuración del TSS.

Con estos cambios realizados, la función `Worker` (por mantener el nombre) solo tendrá que hacer una resta saturada en una unidad de los tiempos de ejecución restantes de todas las tareas en ejecución y la función `Sched`, si los ciclos restantes de ejecución de la última tarea (al estar ordenadas por ciclos restantes, la más cercana a finalizar) es igual a cero, quiere decir que ha finalizado y se puede expulsar del supuesto procesador. Si no se cumple esta condición, se intenta empezar a ejecutar una nueva tarea en un supuesto procesador libre si es posible. Para ello se comprueba si las tareas en ejecución es menor a los procesadores disponibles, si esta condición es cierta se intenta añade de forma ordenada la nueva tarea a la lista de tareas en ejecución.

Como se puede observar en la tabla 17, los tiempos de ejecución se han reducido considerablemente y se alcanzan *SpeedUps* (Tabla 18) cercanos a los deseados al inicio del proyecto.

	Tiempo llchol-trace-2048-32 (s)	Tiempo llchol-trace-2048-64 (s)	Tiempo llchol-trace-2048-128 (s)
Config - 1	13,1	9,63	8,36
Config - 4	7,51	5,67	10,6
Config - 8	4,61	7,81	18,1

Tabla 17: Tiempos de ejecución de la v2.0

	SpeedUp llchol-trace-2048-32	SpeedUp llchol-trace-2048-64	SpeedUp llchol-trace-2048-128
Config - 1	2,16	2,33	2,29
Config - 4	9,45	9,56	11,91
Config - 8	16,30	22,08	27,38

Tabla 18: SpeedUp de la v2.0

4.1.7 SIMD, Bithacks y desenrolle de bucles - v2.1

Para mejorar aún más la versión obtenida, se ha hecho uso de la colección de instrucciones vectoriales SSE4.2, como en una de las versiones anteriores, pero en este caso para realizar la resta saturada sobre los ciclos restantes de cada tarea en ejecución. Esta colección dispone de una instrucción que realiza la resta saturada directamente, pero solo con registros divididos en variables de 8 o 16 bits. En consecuencia se ha utilizado una versión con técnicas de *Bithacks*. Para reducir aún más los overheads provocados por el bucles, se hace un desenrolle de bucles en un nivel. En las siguientes dos tablas (Tabla 19 y Tabla 20) se puede observar los tiempos de ejecución con estos dos cambios y el SpeedUp respecto a la versión original

	Tiempo llchol-trace-2048-32 (s)	Tiempo llchol-trace-2048-64 (s)	Tiempo llchol-trace-2048-128 (s)
Config - 1	12,8	8,89	7,5
Config - 4	7,28	5,46	10,27
Config - 8	4,55	7,52	17,8

Tabla 19: Tiempos de ejecución de la v2.1

	SpeedUp llchol-trace-2048-32	SpeedUp llchol-trace-2048-64	SpeedUp llchol-trace-2048-128
Config - 1	2,21	2,53	2,55
Config - 4	9,75	9,93	12,29
Config - 8	16,51	22,93	27,84

Tabla 20: SpeedUp de la v2.1

4.2 Paralelización

El objetivo de esta fase es habilitar el código de la versión 2.1 del simulador desarrollada en la fase de optimización para poder ejecutar el simulador de forma paralela en diferentes núcleos de un procesador y para ello se decide utilizar OpenMP.

Primero de todo, se ha analizado el código con la ayuda de la herramienta *Tareador* y con los conocimientos obtenidos en la carrera sobre este tipo de programación para realizar satisfactoriamente este tipo de optimización del código.

Para entender mejor las diferentes estrategias de paralelización se explicará el diseño del simulador. Primero de todo se inicializan todas las variables a partir de los parámetros de entrada. Entonces, entra en un bucle donde cada iteración simula ser un ciclo del TaskSuperScalar. En cada iteración del bucle principal hay un bucle por cada tipo de componente, y este bucle interno recorre los diferentes componentes del mismo tipo para simularlos uno a uno, como pueden ser los diferentes procesadores en la versión original. También están los árbitros que manejan el uso de las FIFOs que interconectan los diferentes componentes. En la Figura 8 se puede ver una aproximación muy reducida con pseudocódigo de lo que sería el simulador.

```
init()
Cycle = 0
while(! Fin)

   for(i = 0; i < #Componentes_1; ++i)
      componente_1(i)

   for(i = 0; i < #Componentes_2; ++i)
      componente_2(i)
   …
   Arbitros ( )
   Cycle = Cycle + 1
```

Figura 8: Aproximación diseño del simulador en pseudocódigo

Analizando el código y con la ayuda de *Tareador* se detecta que hay diferentes bucles sin dependencias entre iteraciones de los mismos. Estos bucles son los que recorren los componentes del mismo tipo. Así que se deciden paralelizar las iteraciones de los bucles que recorren los componentes del mismo tipo, pero sin ejecutar simultáneamente iteraciones de diferentes bucles internos del bucle principal, es decir, bucles de componentes diferentes.

Al paralelizar estos bucles se comprueba que no provoca ninguna mejora en los tiempos de ejecución, sino todo lo contrario, aumenta al considerablemente los tiempos de ejecución. La razón por la que no se consigue ninguna mejora con esta primera estrategia de paralelización es debido a que estos bucles están dentro del bucle que representa cada ciclo del simulador. En cada ejecución, de media se simulan decenas de millones de ciclos, es decir hay decenas de millones de iteraciones del bucle principal. El problema reside en que el tiempo de ejecución en una sola iteración de cualquiera de las funciones que consumen más tiempo de ejecución es mínimo. Esto provoca que el *overhead* creado en cada ciclo por la programación paralela sea mayor que el tiempo de ejecución de un ciclo ejecutado de forma secuencial. En la Tabla 21 se pueden observar el incremento en los tiempos de ejecución y en la Tabla 22 como los SpeedUps (calculados respecto a la versión original) caen en picado y arruinan las mejoras obtenidas en la fase de optimización, incluso obteniendo tiempos superiores a la versión original.

	Tiempo llchol-trace-2048-32 (s)	Tiempo llchol-trace-2048-64 (s)	Tiempo llchol-trace-2048-128 (s)
Config - 1	226,03	148,47	131,68
Config - 4	270,25	165,28	396,01
Config - 8	131,68	146,08	358,78

Tabla 21: Tiempos de ejecución de la primera versión paralelizada

	SpeedUp llchol-trace-2048-32	SpeedUp llchol-trace-2048-64	SpeedUp llchol-trace-2048-128
Config - 1	0,12	0,15	0,14
Config - 4	0,26	0,33	0,32
Config - 8	0,96	1,18	1,38

Tabla 22: SpeedUps de la primera versión paralelizada

La segunda estrategia que se plantea es paralelizar componentes diferentes entre sí y dejar en serie las

iteraciones de los bucles que recorren los componentes del mismo tipo. Pero se empeora aún más los tiempos de ejecución. Sigue teniendo el mismo problema, el *overhead* creado en la creación de threads y asignación de trabajo a los diferentes núcleos mata la mejora de rendimiento que se pueda obtener. Se decide descartar la estrategia anterior y crear diferentes threads con diferentes bucles e intentar sincronizarlos como se muestra en la Figura 9. Esta estrategia se basa en que se crea un hilo para cada bucle. Todos los bucles empiezan las iteraciones a la vez y todos los bucles que acaban una iteración esperan al último en finalizar. Para un funcionamiento óptimo de esta estrategia, se ha intentado balancear la carga de los bucles teniendo en cuenta las posibles dependencias y acceso a datos compartidos por parte de las diferentes funciones.

```
init
Cycle = 0
while (! Fin){  // ejecutado en el thread 0
   for(i = 0; i < #Componentes_1; ++i)
      componente_1(i)
   sync
}

while (! Fin){   // ejecutado en el thread 1
   for(i = 0; i < #Componentes_1; ++i)
      componente_2(i)
   sync

…
   Arbitros ( )
   Cycle = Cycle + 1
}
```

Figura 9: Pseudocódigo de prototipo de paralelización

El problema de esta tercera estrategia es que la sincronización y la compartición de memoria provocan un *overhead* que no permite obtener una mejora de rendimiento del simulador. Como se puede observar en la las tablas 23 y 24, esta estrategia ha sido la que ha estado más cerca de conseguir una versión que mejorará el rendimiento del programa, pero sigue empeorando los tiempos respecto al os obtenidos en la fase de optimización

	Tiempo llchol-trace-2048-32 (s)	Tiempo llchol-trace-2048-64 (s)	Tiempo llchol-trace-2048-128 (s)
Config - 1	45,25	33,51	27,7
Config - 4	21,5	15,44	36,47
Config - 8	15,15	34,42	98,53

Tabla 23: Tiempos de ejecución de la tercera versión paralelizada

	SpeedUp llchol-trace-2048-32	SpeedUp llchol-trace-2048-64	SpeedUp llchol-trace-2048-128
Config - 1	0,63	0,67	0,69
Config - 4	3,3	3,51	3,46
Config - 8	4,96	5,01	5,03

Tabla 24: SpeedUps de la tercera versión paralelizada

Finalmente se decide que no compensa el uso de paralelización para el simulador ya que no se ha obtenido ninguna versión que rebaje los tiempos de ejecución. La configuración 1 del TaskSuperscalar es con la que se había obtenido menos mejora de rendimiento en la fase de optimización y por ello se esperaba conseguir una mejora en la fase de paralelización. Esto no ha sido así, sino todo lo contrario, porque en cada ciclo se produce un *overhead* y con esta configuración se ejecutan muchos más ciclos (iteraciones del bucle principal). Esta es la razón de que los *SpeedUps* sean peores con configuraciones que necesiten más ciclos.

4.3 Resultados

La v2.1 es la versión final de la fase de optimización, y por consecuencia de no encontrar una solución con paralelización también es la versión final del proyecto. Con una versión final ya definida, se han hecho ejecuciones con todos los tipos de trazas y tamaños para comparar los *SpeedUps* obtenidos. No se hace una comparación de los tiempos de ejecución porque varían demasiado entre aplicaciones y trazas. En la Figura 11 se observa que para las trazas *heat-trace-2048-256* y *lu-trace-2048-4* se obtienen *SpeedUp* muy parecidos a los obtenidos con las trazas utilizadas para hacer las diferentes pruebas de las secciones anteriores (las llchol-trace-2048-32,64 y 128). La *llchol-trace-2048-8* también se asemeja para las dos configuraciones más pequeñas, pero para la tercera no se obtiene el *SpeedUp* de las trazas anteriores. Esto puede ser debido a que esta traza es la más grande que se ha utilizado (200 Megabytes). La traza que presenta unos resultados más diferentes a los obtenidos anteriormente es la *sparselu-trace-2048-8*, debido a la naturaleza de la aplicación.

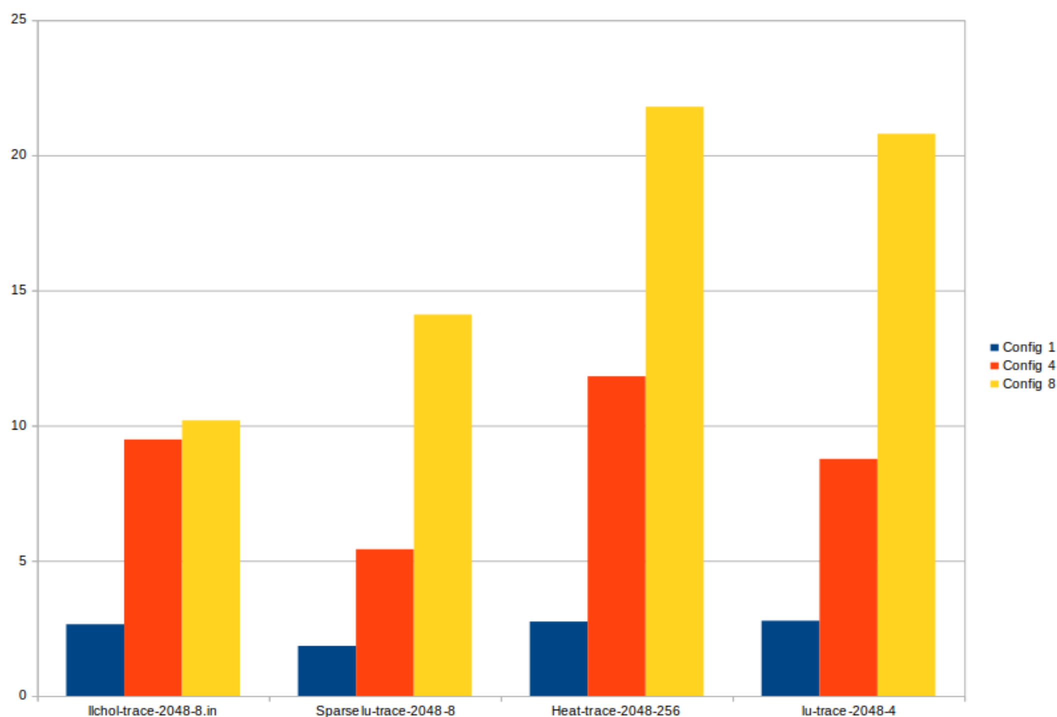

Figura 11: Comparación de SpeedUp con diferentes trazas

La mayor mejora obtenida ha sido gracias a darle un nuevo replanteamiento al código y a sus estructuras de datos, y así eliminar regiones del código que dejaban ver las flaquezas del computador o del código, como predicciones fallidas por parte del computador o no explotar la localidad espacial . En la figura 10 se puede observar la progresión de *SpeedUps* de las diferentes versiones (sin contar las de la fase de paralelización), y como a partir de la mejora hecha replanteando la estructura de datos conflictiva (v2.0) se disparan los *SpeedUps*. Las ejecuciones de la Figura 10 están hechas con la traza "llchol-trace-2048-128".

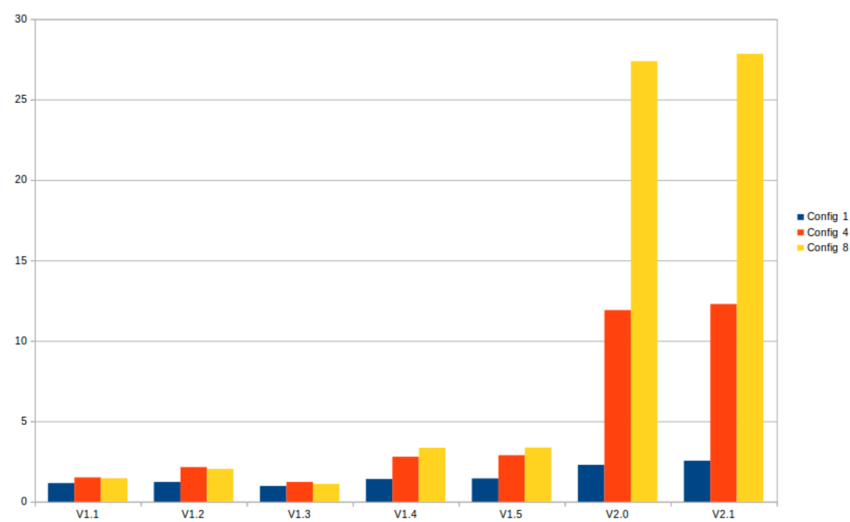

Figura 10: Diferencia de SpeedUp entre versiones

5. Conclusiones

Para realizar un desarrollo de un software eficiente hay que tener en cuenta que hay que focalizar el trabajo en las regiones de código que consumen más tiempo de ejecución. La Ley de Amdahl define que la mejora obtenida en el rendimiento al utilizar algún método de ejecución más rápido está limitada por la fracción de tiempo que se pueda utilizar ese modo más rápido. Es decir, por mucho que se optimice una función que consume un cinco por ciento del tiempo total, no se va a obtener más de un cinco por ciento de mejora. Esto se ha visto claramente en la realización del proyecto. Gracias a focalizar la mayoría de trabajo sobre las dos funciones que consumían en la versión original el 90%, o incluso más, del tiempo de ejecución se ha logrado alcanzar SpeedUps alrededor del 10x o 25x según la configuración.

A partir de los resultados obtenidos, se puede observar que teniendo consciencia de la arquitectura del computador donde se ejecutará un software que se está desarrollando puede llevar al programador a conseguir una versión con un rendimiento mucho mayor. Es cierto que las mejoras hechas a consciencia de la arquitectura del computador (optimización de memoria, uso de SIMD, optimización del control del flujo, etc.) han reducido los tiempos de ejecución, pero la mayor mejora se ha obtenido dándole un nuevo planteamiento al código y definiendo de nuevo las manteniendo la funcionalidad.

Además, la programación paralela no es trivial y se demostrado en este trabajo. Desde el principio del proyecto se pensaba que se podría aplicar una paralelización al código ya que los componentes del hardware que se está simulando deberían trabajar en paralelo por la naturaleza del hardware. Finalmente no ha sido posible por el coste que supone este tipo de programación en la compartición de datos por parte de los diferentes núcleos y la creación y asignación de hilos. La única paralelización obtenida ha sido el uso de instrucciones vectoriales para trabajar de forma paralela con diferentes operandos simultáneamente.

6. Trabajo futuro

Uno de los problemas encontrados en la realización del proyecto ha sido la falta de la colección de instrucciones vectoriales AVX2. Con esta colección se hubieran podido utilizar el doble de operandos simultáneamente en las regiones de código donde se ha utilizado instrucciones SIMD. Si en un futuro se decide optimizar aún más el simulador y se dispone de un procesador con la colección AVX2, la adaptación del código actual sería sencilla. Las instrucciones vectoriales serían las mismas pero con 256 bits, y se debería tener cuidado de tratar con los índices correctos de los vectores.

Además, actualmente Intel está desarrollando procesadores con una nueva colección de instrucciones que trabajaran con 512 bits o incluso 1024 bits. Los procesadores que tienen soporte para este tipo de instrucciones son los que están en la generación “Knights Landing” de Intel, como el Intel Xeon Phi. Estos procesadores tienen un precio desorbitado, pero en un futuro, cuando ya estén trabajando en nuevas arquitecturas, los precios bajaran y serán más asequibles.

Ya que no se ha encontrado una solución multi-hilo para el código del simulador, y con las futuras novedades en las instrucciones vectoriales se podría adaptar totalmente el código para hacer uso de estas instrucciones.

7. Glosario

Árbitros: Componentes del TSS y el SimTSS que mantienen el orden en las interconexiones de los componentes mediante FIFOs.

Bithacks: Técnica de programación que trata las variables as consciencia de cómo se almacenan y representan en memoria.

Inlining: Es una técnica de optimización que reemplaza en el código la llamada a una función por el código de la misma función directamente.

OpenMP: Es una API que da soporte a la programación multiproceso de memoria compartida.

Profiling (Profiler): Análisis de rendimiento de un programa cualquiera mediante herramientas que utilizan diferentes métodos para obtener diferentes resultados. Las herramientas que se utilizan son los Profilers.

Sched: Función del SimTSS que expulsa las tareas finalizadas de los procesadores o si hay procesadores libres les asigna una tarea.

SpeedUp: Es utilizado para calcular las mejoras de rendimiento realizadas. En este caso se define como la división entre el tiempo de ejecución original y el tiempo de ejecución de una versión modificada.

Tareador: Herramienta para el desarrollo de aplicaciones paralelas que permite analizar las dependencias entre tareas definidas por el programador.

Worker: Estructura de datos del SimTSS que emula y define los procesadores en los cuales se ejecutarán las tareas definidas por las trazas.

Workers: Función del SimTSS. En la versión original comprueba si una tarea en ejecución ha finalizado. En la versión final hace una resta saturada en una unidad a los ciclos restantes de ejecución de las tareas en ejecución.

8. Abreviaturas

API - Interfaz de Programación de aplicaciones

AVX – Extensiones vectoriales avanzadas

CPD - Centro de Procesamiento de Datos

CPU – Unidad central de procesamiento

FGPA – Field-Programmable gate array

FIFO – First In, First Out (

GPL – General Public License

PC – Computadora Personal

SIMD – Single Instruction, Multiple data

SimTSS – Simulador del TaskSuperscalar

SoC – System-on-Chip

SSE – Streaming SIMD Extensions

TSS – TaskSuperscalar

9. Bibliografía

[1] F. Yazdanpanah, “Hardware Design of Task Superscalar Architecture,” Universitat Politècnica de Catalunya, 2014.

[2] F. Yazdanpanah, D. Jimenez-Gonzalez, C. Alvarez-Martinez, Y. Etsion, and R. M. Badia, “Analysis of the task superscalar architecture hardware design,” *Procedia Comput. Sci.*, vol. 18, pp. 339–348, 2013.

[3] F. Yazdanpanah, D. Jimenez-gonzalez, C. Alvarez-martinez, Y. Etsion, and R. M. Badia, “FPGA-Based Prototyp e of the Task Sup erscalar Architecture,” Belin, Germany, 2013.

[4] M. Goll and S. Gueron, “Vectorization on ChaCha stream cipher,” in *ITNG 2014 - Proceedings of the 11th International Conference on Information Technology: New Generations*, 2014, pp. 612–615.

[5] *OpenMP application program interface version 4.0*. [S.l.] : OpenMP, 2013.

[6] T. G. Mattson, B. A. Sanders, and B. Massingill, *Patterns for parallel programming*. Boston [etc.] : Addison-Wesley, 2005.

[7] A. Grama, *Introduction to parallel computing*. Harlow, England : Pearson Education, 2003.

[8] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” *IEEE Comput. Sci. Eng.*, vol. 5, 1998.

[9] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, vol. 10. 2008, p. 353.

[10] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*, vol. 129. 2001, p. 230.

[11] R. E. Bryant and D. O’Hallaron, *Computer systems : a programmer’s perspective*. Boston [etc.] : Pearson, 2011.

[12] Intel, “Intel SSE4 Programming Reference,” *Integers*, 2007.

[13] I. Corporation, “Intel ® 64 and IA-32 Architectures Software Developer ’ s Manual Volume 3A : System Programming Guide , Part 1,” *System*, vol. 3, pp. 1807–1814, 2010.

[14] D. Rother, “Using AVX2 for the Optimization of a Software-based Real-Time LDPC Decoder,” in *8th Karlsruhe Workshop on Software Radios*, 2014, pp. 1–8.