INSTITUTO DE COMPUTACIÓN - FACULTAD DE INGENIERÍA

SimCo - Un simulador de procesadores con enfoque educativo

Federico Rivero

Tutores: Eduardo Grampín, Matías Richart

29 de julio de 2014

1. Resumen

La técnica de simulación en el área de arquitectura de computadoras es usada a nivel académico fundamentalmente para evaluar nuevos diseños. Es un ambiente en el que resulta natural utilizar esta metodología, puesto que es impracticable la exploración del espacio de diseño mediante la construcción de prototipos, por el alto costo de los mismos. Por otro lado, la industria de microprocesadores lucha cuesta arriba con el problema práctico de que el tiempo que transcurre entre que se realiza el diseño de un nuevo microprocesador y este resulta fabricado, resulta demasiado largo como para ser aplicado en todos los casos, dando lugar a que la simulación sea también una estrategia atractiva para la evaluación de rendimiento de nuevos procesadores en este ambiente. Por último, otro lugar natural para la simulación es el ámbito educativo, pues además de que resulta prohibitivo económicamente disponer de todas las piezas de hardware que se desearían analizar, es evidente que no se puede realizar un análisis observacional práctico de qué sucede en un microprocesador durante la ejecución de un cierto programa.

En otro punto, la construcción de procesadores más veloces se encuentra estancada desde principios de los años 2000 debido a los problemas impuestos por la barrera física de la velocidad de la luz. Dado esta situación, el aumento en rendimiento de los procesadores y por tanto su diseño actual está enfocado en explotar el llamado paralelismo a nivel de hilo, mediante la ejecución paralela de varios hilos de ejecución en un mismo CPU (procesadores denominados *multihilo*) o mediante la disposición de varios núcleos de ejecución integrados en el mismo chip (procesadores *multinúcleo*). Esto vuelve a dichos procesadores elementos de interés a nivel académico, pero su estudio aún no ha sido abordado por el grupo MINA del Instituto de Computación de la Facultad de Ingeniería.

Dada la realidad presentada anteriormente, el objetivo del proyecto es realizar una actualización en la materia de simulación en arquitecturas de computadoras, encontrar espacio abierto de investigación en el área y finalmente desarrollar o extender un simulador con enfoque educativo y con soporte de simulación para procesadores multihilo y multinúcleo, para uso en las asignaturas del Departamento de Arquitectura, Sistemas Operativos y Redes de Computadoras del Instituto de Computación, Facultad de Ingeniería.

ÍNDICE

2. Introducción

El objetivo de la simulación de procesadores es brindar, dado un programa y la especificación de un cierto procesador que lo pueda ejecutar, una descripción lo más acertada posible de qué sucedería si se ejecutara dicho programa en ese procesador. A pesar de que la explicación de las funcionalidades de este tipo de simuladores se resume a un único caso de uso, estos son piezas de software sumamente complejas, no por la simulación en si, sino porque los procesadores actuales son complicados. Por esta razón, el desarrollo de un simulador de procesadores conlleva un proceso largo de desarrollo, provocando que no abunden demasiado. Los simuladores usados a nivel industrial son propietarios y por tanto no están disponibles para ser usados a nivel académico, sin embargo existen varios simuladores de código abierto [13] [17] [10].

Para poder comprender el funcionamiento de los simuladores de procesadores usados en la actualidad, se deben abordar varias temáticas que no son estudiadas en la carrera de grado de Ingeniería en Computación, por lo tanto, en el resto de esta sección se realizará una introducción a algunos temas abarcados a lo largo del proyecto, con los cuales en su mayoría no se espera que el lector esté familiarizado.

2.1. Arquitectura y Microarquitectura de Computadoras

Dado que durante este trabajo se realiza un trabajo muy íntimo con la arquitectura y microarquitectura de las computadoras, es conveniente comenzar definiéndolas. La arquitectura de una computadora (o su más descriptivo nombre en inglés: *Instruction Set Architecture - ISA*), refiere a aquellos aspectos de la computadora que son visibles al programador, mientras que la microarquitectura concierne a cómo esta se implementa. Dentro de la arquitectura de una computadora quedan incluidos elementos tales como el set de instrucciones del procesador y qué registros programables contiene, mientras que dentro de la microarquitectura se incluyen aspectos tales como la disposición de los componentes, qué frecuencia de reloj se utiliza o qué tecnología de fabricación de semiconductores es utilizada.

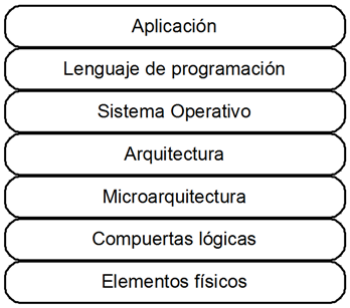


Figura 2.1: Modelo en capas de un sistema computacional

Si se realiza una abstracción por capas de la estructura lógica utilizada en la programación y ejecución de un programa, se explicita que todo lo que queda por debajo de la arquitectura refiere a aspectos de hardware, mientras que aquellas capas superiores corresponden a piezas de software. Por esta razón es que la arquitectura de computadoras es referida como la *interfaz entre el hardware y el software*, denominación que será utilizada a lo largo del trabajo pues es más apropiado que el término arquitectura de computadoras, dado que el objetivo del proyecto no es simular totalmente una *computadora* según la definición más común que se tiene de ella [4], sino limitarse a los procesadores y su interacción con el sistema de memoria.

2.2. Simulación en Arquitectura de Computadoras

Los simuladores de procesadores se clasifican primordialmente en simuladores funcionales y de rendimiento [9]. Los primeros implementan únicamente la arquitectura del procesador simulado, es decir que brindan el estado del procesador y la memoria luego de cada instrucción, pero no indican *cómo* se llegó a ese estado. Por el contrario, los simuladores de rendimiento indican también con mayor o menor exactitud, qué es lo que ocurre a nivel circuital, es decir que también simulan la microarquitectura del procesador, pudiendo de este modo responder preguntas como cuánto tiempo duraría la ejecución, qué recursos serían utilizados y con qué frecuencia, cuánto calor sería disipado por el circuito, etc. Los simuladores funcionales son intrínsecamente más sencillos de implementar que los de rendimiento, pues del estado del circuito se puede deducir el estado del sistema computacional, siendo evidente que todo simulador de rendimiento es también un simulador funcional. De todos modos, esto no vuelve a estos últimos inútiles, pues la sencillez de su implementación provoca que sean de más rápida ejecución.

Dado que los simuladores de rendimiento se concentran en la microarquitectura, se debe brindar una descripción de la misma como entrada del simulador. La clasificación de estos simuladores se puede realizar según cuál sea el nivel de detalle de la simulación. Típicamente un simulador de rendimiento simula como mínimo el estado del hardware ciclo por ciclo. Estos son llamados simuladores de precisión de ciclo (cycle-accurate).

Como se verá a lo largo de este trabajo, es posible simular con más precisión que a nivel de ciclo, obteniendo resultados intermedios. A ese tipo se los llamará *simuladores de precisión de subciclo*. Por último, a un simulador que simule fielmente el hardware de un procesador, calculando en software el valor de las señales físicas y su propagación a través de las compuertas lógicas se lo llamará *simulador de señales*. La decisión de qué simulador utilizar o desarrollar se debe realizar en función de dos parámetros esenciales, generalmente contrapuestos: el nivel de detalle que se desea obtener y la velocidad a la que se desea simular. Existen situaciones en las cuales uno de los dos parámetros no es relevante, como por ejemplo al simular un programa muy corto, o si directamente un cierto nivel de detalle ya no es relevante, pero en general la decisión dependerá de la ponderación de ambas

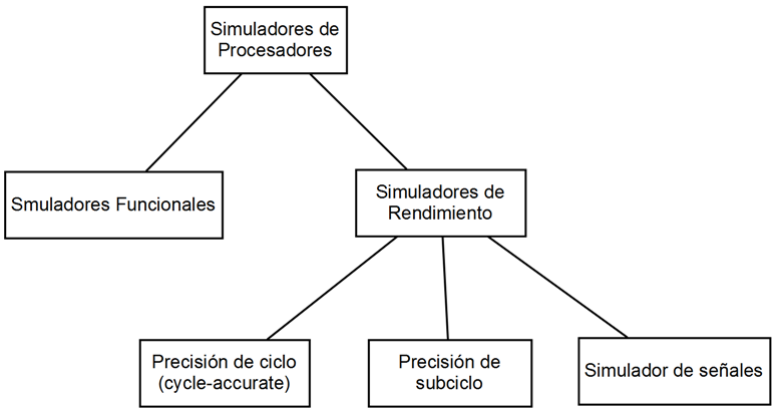


Figura 2.2: Clasificación de simuladores de procesadores en base a su nivel de detalle

variables.

Adicionalmente a la división en función del nivel de detalle, los simuladores también se clasifican según la entrada que obtienen para realizar la simulación. Una primera opción es obtener resultados ejecutando el programa en el procesador simulado, a este tipo de simuladores se los denomina guiados por ejecución (*execution-driven*). Para este tipo de simuladores, la entrada es un programa en algún formato, algunos ejecutan programas directamente en binario mientras que otros toman programas escritos en ensamblador o incluso en lenguajes de alto nivel. La segunda opción consiste en tomar como entrada del simulador una traza de la ejecución del programa (referencias a memoria, valores guardados en registros, resultados de predicción de saltos, etc) y a partir de ellos reproducir la ejecución en el hardware simulado, estos últimos se denominan guiados por trazas (*trace-driven*).

El rol de la simulación en el proceso de investigación en arquitectura de computados ha ganado importancia con el tiempo y se ha consolidado como el medio por excelencia para la evaluación de rendimiento de nuevos diseños. En la figura 2.3 se muestra, para algunos años representativos, la evolución de la proporción de artículos presentados en la conferencia HPCA (IEEE International Symposium On High Performance Computer Architecture) [2] que utilizan la simulación como método de validación de los estudios (datos tomados de [9]).

Más información al respecto del rol de la simulación para la arquitectura de computadoras puede encontrarse en [9] y [11].

2.3. Procesadores Multinúcleo

Uno de los objetivos principales del proyecto es el de soportar la simulación de multiprocesadores. Por esta razón se realizará en esta sección una breve introducción a las características

Año	Artículos totales	Con simulación
1973	28	2
1985	43	12
1993	32	23
1997	30	24
2001	25	22
2004	31	27

Figura 2.3: Rol de la simulación en conferencia HPCA

más importantes de este tipo de procesadores, los cuales no son abordados en la currícula básica del ingeniero en computación. A nivel lógico (es decir, dejando de lado consideraciones físicas), un multiprocesador consiste en múltiples CPU's, conectadas por una cierta red y operando con un cierto conjunto de memoria, generalmente compartida. Cuando dichos procesadores se encuentran dispuestos en un mismo chip, se denominan procesadores *multinúcleo (multicore)*, y a la red que los interconecta se la denomina Network on Chip (NoC). Se diferencian de una red de computadoras tradicional en que en la mayoría de los casos la interconexión no se realiza a través del subsistema de entrada/salida.

2.3.1. Esquemas de memoria compartida

La relación de los multiprocesadores con la memoria se puede clasificar en dos: de memoria compartida o de memoria no compartida. Aquellos con memoria compartida disponen de un espacio de direccionamiento común y se los puede subclasificar según su tiempo de acceso a la memoria: si el tiempo de acceso es común para todas las direcciones de memoria se dice que el procesador es *UMA (Uniform Memory Access - Accesso Uniforme a memoria)*. La figura 2.4 muestra un multiprocesador de memoria compartida UMA.

Si en cambio el tiempo de acceso difiere según la dirección accedida, se trata de un procesador *NUMA (Non Uniform Memory Access - Acceso No Uniforme a Memoria)*. La figura 2.5 muestra la organización de un multiprocesador de memoria compartida NUMA. Curiosamente, esa figura también puede representar la organización de un multiprocesador de memoria no compartida, que son aquellos donde el espacio de direccionamiento físico es diferente para algunos o todos los núcleos de ejecución (es decir, escriben en lugares físicos de memoria diferentes). A este tipo de multiprocesadores también se los denomina *message-passing multiprocessors* (multiprocesadores de traspaso de mensajes), pues la comunicación de los procesadores se realiza justamente a través de mensajes y no de posiciones de memoria común.

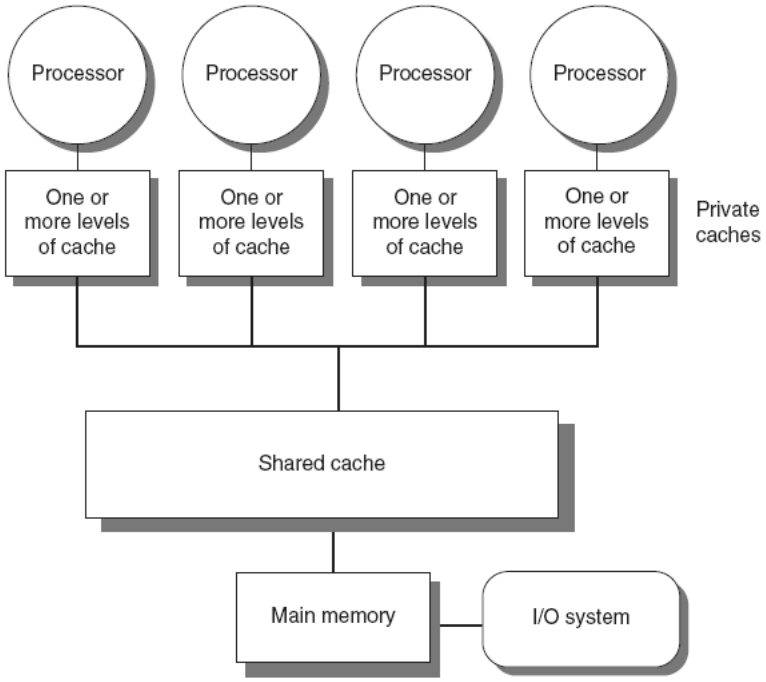


Figura 2.4: Procesador de memoria compartida UMA - *Imagen tomada de [1]*

2.3.2. TIPOS DE REDES DE INTERCONEXIÓN

En [5] se define una red de interconexión como *un sistema programable que transporta datos entre terminales*. A lo largo de este trabajo se abordarán dos tipos de redes de interconexión diferentes: buses y redes basadas en switches, aunque de los dos, mayoritariamente se utilizará el medio de difusión clásico de las computadoras: el bus. Se dará por sentado que el lector conoce las características elementales de este elemento de la arquitectura Von Neumann.

Se denomina dominio de difusión al conjunto de elementos que deben recibir un mensaje de difusión en un medio compartido. El CPU, la memoria y los diferentes dispositivos de E/S forman el dominio de difusión clásico de la arquitectura Von Neumann. A medida que el número de núcleos del PC aumenta, también lo hace el dominio de difusión, aumentando superlinealmente el ancho de banda requerido por el medio compartido, y en el caso de que éste sea implementado a través de un bus, complejizando seriamente su diseño por cuestiones de electrónica en las cuales no se entrará en detalle. Ver [5] por una explicación al respecto.

La realidad presentada anteriormente llevó a los investigadores a proponer diseños alternativos al bus para los multiprocesadores con un número de núcleos elevado. Para un número de núcleos mayor a 32 [1], se utilizan interconexiones similares a las de las redes de computadoras, consistentes en enlaces punto a punto y conmutadores (switches). Si

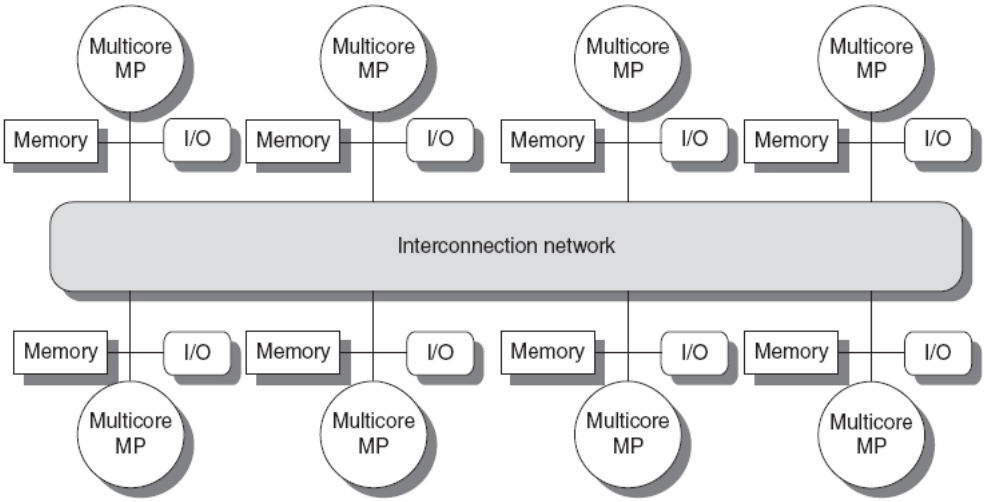


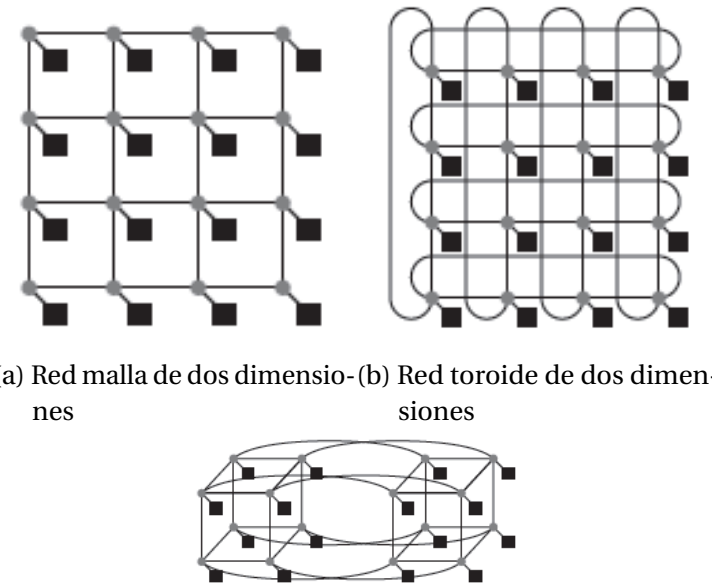
Figura 2.5: Procesador de memoria distribuida (compartida o no compartida)
Imagen tomada de [1]

la topología de la red es buena, se logra proveer un ancho de banda superior que con un bus. La contrapartida de este beneficio es que se deben resolver los problemas tradicionales de redes, como el enrutamiento y el reenvío, la pérdida de paquetes por sobrecarga en los conmutadores, entre otras. Estas complejidades serán abordadas superficialmente a lo largo del presente trabajo.

La topología juega un papel importante en el ancho de banda provisto por la red. En la figura 2.6 se muestran tres de las topologías usadas en multiprocesadores comerciales, cada nodo de la red es un conmutador, el cual típicamente está conectado a un único dispositivo, ya sea un procesador o de memoria. Dado que en estas redes el enrutamiento se realiza generalmente de forma estática, cada topología utiliza una numeración y algoritmo de enrutamiento diferente. Por ejemplo, para una red de malla, el algoritmo usado por lo general es llamado enrutamiento por dimensión (*dimension order routing*), en el cual primero se realiza el reenvío en una dimensión hasta que esta coincide con la del destino y luego en la segunda dirección. Por ejemplo, si se debe enviar un paquete desde el nodo superior izquierdo hasta el inferior derecho y se utiliza este algoritmo con orden de dimensiones horizontal primero, vertical después, el paquete recorrerá la fila superior y al llegar al nodo superior derecho será reenviado por la columna de más a la derecha, hasta que llegue a su destino.

2.3.3. Arbitraje en NoCs

Se denomina arbitraje a la función de determinar a qué dispositivo se le permite acceso a un medio compartido en un determinado tiempo. Si bien en la arquitectura Von Neumann



(a) Red malla de dos dimensiones

(b) Red toroide de dos dimensiones

(c) Red hipercubo

Figura 2.6: Algunas topologías de redes de interconexión

tanto el CPU como los dispositivos de entrada salida deben hacer uso del bus, durante la mayoría del tiempo es el CPU quien lo controla. Al agregar otros procesadores al dominio de difusión, se aumenta la probabilidad de que estos deseen hacer uso del medio compartido al mismo tiempo. La función de arbitraje en estos casos cobra mayor importancia.

Funciones de arbitraje clásicas de los buses son *round robin*, la cual consiste en asignar el control del bus a los distintos dispositivos en forma equitativa en el tiempo, y *least recently served*, en el cual se asigna control del bus a aquel dispositivo que hace mayor tiempo realizó su última transferencia.

2.3.4. Coherencia de cache

El uso de caches en procesadores multinúcleo UMA acentúa el problema de coherencia de cache que aparece en monoprocesadores cuando operan con un controlador DMA [1]. Para solucionar este problema, se implementa algún denominado protocolo de coherencia de cache, encargado de mantener actualizados los diferentes valores guardados en los dispositivos de memoria. Estos protocolos se clasifican en dos grupos, los del primer grupo

[1] Se denomina problema de coherencia de cache, cuando por algún motivo el valor contenido en un cache no es el correcto según el sistema de memoria. Por ejemplo, si un controlador DMA escribe un valor en una dirección de memoria principal, la cual estaba guardada también en memoria cache, si no se toma ninguna acción y el procesador realiza una lectura a dicha dirección, obtendrá el valor desactualizado desde el cache.

se denominan protocolos de husmeo (*snooping protocols*), son usados fundamentalmente en procesadores simétricos (ver figura 2.4) y consisten en que los caches de último nivel (*last level caches* - aquellos que están más cerca de la memoria principal) estén pendientes de las transferencias realizadas en el bus principal, de forma tal que al notar un pedido de lectura o escritura a una dirección contenida en el cache, se tomen las acciones necesarias para que no ocurra un problema de coherencia (por ejemplo, si se realiza una escritura en el bus principal a una dirección que está en un cache, dicho cache puede actualizar el valor guardado con el valor disponible en el bus). Los protocolos del segundo grupo se denominan protocolos basados en directorios (directory based protocols), son utilizados generalmente en sistemas de memoria distribuida (ver figura 2.5) y consisten en disponer de forma centralizada el estado de cada bloque de memoria, en estructuras de hardware llamadas directorios. De este modo, cuando se quiere realizar una lectura o escritura sobre una línea de cache, se envía un mensaje al directorio indicando la acción realizada, para que este tome las medidas necesarias para mantener el sistema de memoria consistente. Por ejemplo, si un cache de algún procesador contiene el valor de una cierta dirección de memoria y otro procesador quiere escribir en la misma dirección, al escribir el valor en su cache, se le avisará de la escritura al directorio correspondiente, quien a su vez enviará un mensaje al otro cache para o bien actualizar el valor o para invalidar la línea (según el protocolo utilizado).

En este proyecto se trabaja con ambos tipos, a nivel de husmeo se utilizan los protocolos MSI (utilizado en la bibliografía a modo educativo) y MESI (implementado actualmente a nivel comercial). En el apéndice se encuentra una descripción de dichos protocolos.

3. Estado del Arte en Simulación de Procesadores

En esta sección se describirá el trabajo realizado como relevo del estado del arte en el área de simulación de procesadores.

La investigación se realizó en base a lectura de artículos científicos (papers) al respecto de simulación en arquitectura de computadores, pues en general cada simulador estudiado incluía en su preámbulo un resumen del estado del arte al tiempo de su implementación. Como punto de partida para la búsqueda se tomó aquellos simuladores recomendados en la bibliografía de facto [1] [5]

Además se encontró que muchos artículos en los cuales se proponían nuevas técnicas o diseños para procesadores validaban sus resultados usando algún simulador, por tanto, la otra fuente de datos para el estado del arte fueron los artículos presentados en las últimas conferencias internacionales en la materia de arquitectura, microarquitectura y diseño de computadores [2], [3] [6] [7] [8].

3.1. Reseña de simuladores

El proceso de estudio y evaluación de los diferentes simuladores se trató de un punto de complejidad alta en el proyecto, pues como es advertido en prácticamente *todos* los artículos de presentación de simuladores, el proceso de instalación, comprensión, configuración y ejecución de un simulador es sumamente largo y difícil. Por esta razón, el proceso de estudio incluyó dos fases: en la primera se realizó un refinamiento del conjunto de simuladores encontrado, basado en las características mencionadas en los artículos de presentación, así como en la experiencia y comentarios de aquellos trabajos que utilizan los simuladores. Una vez reducido el espectro, la segunda fase de estudio consistió en analizar el código de estos simuladores. Esta última tarea resultó ser muy compleja, conllevando en promedio dos semanas de estudio por cada simulador.

A continuación se presenta una descripción de los simuladores estudiados:

1. SimpleScalar

   El SimpleScalar es un simulador de procesadores desarrollado por Todd Austin en la Universidad de Michigan [10]. Es un simulador de rendimiento, *cycle-accurate*, que permite el modelado de un procesador superescalar fuera de orden y un sistema de memoria clásico (niveles L1 y L2 de cache, traducción de memoria virtual y memoria física). Es un simulador guiado por ejecución el cual toma como entrada un programa en lenguaje ensamblador de una arquitectura propia, derivada de MIPS IV [12]. La

distribución incluye además varias versiones más simples del simulador, en las cuales se gana mayor velocidad de ejecución a cambio de menos funcionalidades.

SimpleScalar fue considerado un estándar de facto a principios del 2000 por su enorme popularidad en la comunidad científica y la evidencia de esto es innegable. Como se informa en su página 'Who's using it?', más de un tercio de los artículos en diseño de computadoras publicados en el 2002 validaban sus resultados en SimpleScalar. Otro de los reconocimientos que se debe hacer al SimpleScalar es que fue uno de los primeros simuladores de procesadores y de los que más influyeron en la orientación de la comunidad hacia el uso de estos para evaluar el rendimiento de nuevos diseños, como se puede ver en en la figura 2.3.

Como puntos fuertes del SimpleScalar se pueden destacar lo sencillo de la implementación y su velocidad de simulación de algunos millones de instrucciones por segundo en sus versiones menos detalladas (MIPS). Como puntos débiles se encuentran que no modela organizaciones multinúcleo y que la arquitectura simulada es estática. Si bien existen varios proyectos que extienden las funcionalidades del SimpleScalar, no se encontró que alguno tuviera demasiada relevancia en la comunidad científica.

2. Graphite

   El simulador Graphite fue desarrollado por un grupo de investigadores del MIT y presentado en el 2010 en la conferencia HPCA [13]. Es un simulador de rendimiento de precisión de ciclo, presentado a través de su característica más distintiva, la cual es ser distribuido. Graphite mantiene múltiples hilos de ejecución, manteniendo en cada uno la simulación de una entidad denominada 'tile', la cual consiste en un CPU, algunos caches privados, controlador de memoria y un switch de red para realizar la comunicación entre procesadores.

   Si bien Graphite parece muy prometedor por la proclama de estar desarrollado modularmente, tiene la desventaja de ser sumamente nuevo y por tanto no disponer de demasiadas herramientas.

3. SESC y ESESC

   SESC [29] y ESESC [17] son dos simuladores desarrollados por el grupo de i-acoma de la University of Illinois, siendo el segundo un modificación del primero (E viene de Enhaced). Es un simulador de rendimiento con precisión de ciclo, modular, el cual implementa multiprocesadores con variadas interconexiones y configuraciones de memoria. Lo más atractivo de SESC es la filosofía con la cual fue implementado, ya que la justificación del grupo para la implementación de un nuevo simulador fue crear uno con buena documentación, de forma tal que fuera fácilmente extensible, como se puede ver en el siguiente pasaje tomado del artículo de presentación de SESC: "The biggest challenge for new students in architecture research groups is not passing

theory or software classes. It is not finding a new apartment or registering with the INS. It is understanding the architecture of the processor simulator that will soon confront them. A simulator coded not for perfection, but for deadlines. Even the most well-conceived simulator can quickly look like a Big Ball of Mud to the unitiated."[29].

ESESC es un simulador basado en SESC, el cual tiene como objetivo principal lograr simulaciones rápidas, haciendo uso de una simulación estadística (ver las conclusiones del estudio para una explicación de esta técnica). Si bien en su artículo de presentación parece muy prometedor y seguramente lo sea, al estudiarlo se encontró que los agregados no aportan demasiado para los intereses del proyecto, ya que el aumento de velocidad de ejecución está apuntado a la comunidad científica, que tiene necesidad de realizar extensas y reiteradas simulaciones sobre varias configuraciones de diseño diferentes, a fin de explorar los beneficios de cada una.

4. gem5

   gem5 [39] es un simulador de sistema completo [2] desarrollado por una colectividad científica de diferentes universidades y empresas, con el objetivo de obtener un simulador de amplio espectro, es decir que sea sumamente flexible, con el fin de ser usado para diferentes aplicaciones.

   Según lo reportado en su artículo de presentación, gem5 soporta múltiples set de instrucciones y modelos de CPU, así como posibilidades de jerarquía de memoria e interconexión, volviendo la capacidad de simulación de gem5 la mayor de las encontradas. La gran desventaja de gem5 es su gran número de dependencias, debido a que está basado en otras dos herramientas de simulación, M5 [40] y GEMS [41], lo cual reduce en mucho su portabilidad y vuelve el proceso de instalación largo y complejo.

5. Otros simuladores

   Adicionalmente a los citados anteriormente, durante la primera fase se relevó la existencia de muchos otros simuladores, de los cuales se hizo un estudio más bien superficial. Entre ellos: Zesto [34], MARSS [35], Simics [36], RSim [37] y SimFlex [38].

3.2. Conclusiones del estudio de estado del arte

Del estudio realizado, se percibió un interés generalizado en la comunidad científica por desarrollar más y mejores simuladores de multicores y GPUs. Si bien los simuladores de

[2] Se denominan simuladores de sistema completo (*full sistem simulators*), a aquellos simuladores que implementan un sistema computacional completo, de forma tal que programas reales puedan ejecutar sobre ellos sin modificaciones, incluyendo sistemas operativos.

GPUs no fueron incluidos en la reseña por encontrarse fuera del alcance del proyecto, se encontraron varios artículos presentando estos simuladores [14] [15]. A nivel de simulación de multicores, la preocupación más notoria se encuentra en lograr que las simulaciones sean más veloces. Esto es lógico si se piensa que a mayor cantidad de núcleos que contenga el sistema computacional a simular, mayor es el esfuerzo que se debe realizar para simularlo. Si se toma como referencia el SimpleScalar [10], el cual es un procesador *cycle-accurate* y ejecuta cientos de KIPS (en el orden de un millón de veces más lento que lo que ejecutaría la máquina simulada), resulta presumible que la simulación de un procesador con mil núcleos sea mil veces más lenta [3] logrando una velocidad de simulación de algunas instrucciones por segundo, algo totalmente inaceptable. Varias técnicas se han implementado para afrontar esta problemática: simuladores paralelizables [13]; simulación estadística [18] [17], la cual consiste en hacer varias simulaciones de pequeñas porciones del programa y tomar los resultados como generales, basándose en el teorema central del límite. Para poder realizar una simulación real de una porción de código cualquiera primero se debe aplicar la técnica de *checkpointing*, la cual consiste en realizar una simulación funcional del programa, dejando como salida una traza del estado del sistema computacional ciclo por ciclo y luego realizar las simulaciones de pequeñas porciones partiendo de algún punto en particular (checkpoint); o simulación acelerada por FPGA [19], la cual consiste en programar el análisis temporal en hardware reconfigurable (FPGA).

Por otra parte, se apreció que en los últimos años han aflorado estudios de investigación en arquitecturas heterogéneas, llamadas APU (accelerated processing units), las cuales integran CPU y GPGPU en la el mismo chip [16], delegando las secciones de código mayormente paralelizables para que sea ejecutada en la GPGPU mientras que la CPU ejecuta las funciones del sistema operativo y otras porciones de código más irregulares. A pesar de este interés, no se encontraron simuladores orientados a modelar este tipo de organizaciones, o siquiera alguno que aclamara poder hacerlo. Este es un posible punto de interés para futuros proyectos.

Otro tema activo de investigación es el del control térmico y energético de los componentes computacionales. Se encontró un gran número de artículos orientados a disminuir el consumo de procesadores, memorias y demás, los cuales también basaban sus resultados en simulaciones (ver [20] como ejemplo). Conjuntamente con esto se encontró que muchos simuladores han agregado en los últimos años funcionalidades que permiten modelar componentes físicos de los procesadores (por ejemplo [17]), como ser la tecnología de fabricación, material de construcción, velocidad del ventilador, temperatura ambiente y unas cuantas otras variables, que permitieron determinar a simple vista la complejidad de los modelos implementados.

[3] Más adelante se verá que el orden de ejecución de un simulador es generalmente superlineal en el número de procesadores, por lo cual la suposición anterior es en el mejor de los casos una cota superior de la velocidad de simulación

3.3. EVALUACIÓN DE TRABAJO FUTURO

Una vez culminado el estudio del estado del arte, se estuvo en condiciones de evaluar cómo debía continuar el proyecto. Dicha tarea consistió en decidir si se iba a desarrollar un nuevo simulador o si se iba a extender alguno de los vistos durante el estudio previo.

Realizar una comparación de los simuladores es complejo pues hay muchas características deseables que no todos cumplen, por esta razón, se elaboró un cuadro comparativo con las características deseables del simulador elegido, el cual ayudará para definir una métrica. Para cada característica, se utilizó un rango del 1 al 5 para cuantificar, entendiendo un 1 como que no cumple con la característica y 5 como que la cumple muy bien (por ejemplo, en el ítem 'Configurabiidad de la jerarquía de memoria' se le asignó un 3 al SimpleScalar, pues si bien modela caches, sólo permite configurar uno o dos niveles de jerarquía).

Funcionalidad	SimpleScalar	SESC	ESESC	Graphite	gem5
Detalle ciclo por ciclo (cycle-accurate)	5	5	2	2	2
Simulación de arquitecturas multi-núcleo	1	5	5	4	4
Simulación de procesadores super-escalares	5	5	5	5	5
Configurabilidad de la jerarquía de memoria	3	5	5	4	5
Configurabilidad de la arquitectura	1	1	1	1	4
Buena performance	3	3	5	5	2
Amigabilidad del código fuente	5	3	2	1	1
Documentación disponible	4	5	2	2	3

Adicionalmente a las funcionalidades presentadas en la tabla anterior, se tuvieron en cuenta las características más deseables a obtener al final del proyecto:

- Posibilidad de configuración del set de instrucciones. La funcionalidad de análisis ciclo por ciclo es interesante para cursos avanzados de arquitectura de computadoras, así como para exploración de diseños en futuros proyectos de grado, pero la habilidad de poder incorporar nuevas arquitecturas al simulador es mucho más importante ya que permite la utilización del simulador en cursos donde interese conocer la ejecución funcional de programas en cualquier lenguaje ensamblador.
- Pocas dependencias y requerimientos: uno de las dificultades más grandes halladas durante el estudio de los demás simuladores fue lo complejo del proceso de instalación de los simuladores, los cuales en muchos casos dependían de aplicaciones o bibliotecas no estándar, complejizando el proceso de instalación. Esto va en contra del deseo de simpleza necesario para que el simulador sea utilizado con fines educativos sin demasiada exigencia.

- Es importante que se domine completamente el código del simulador, para de este modo poder asesorar correctamente las extensiones al proyecto, tanto propias como por parte de otros estudiantes de grado. Además, también es importante el dominio a nivel de usuario, para poder facilitar la enseñanza del uso del mismo.

Del análisis realizado se eligió en primera instancia a SESC como simulador a extender, por la gran cantidad de funcionalidades que brinda y porque la filosofía con la que está escrito está alineada con la filosofía del proyecto. Entre todos los simuladores estudiados, realizando un análisis informal parece ser aquel con el cual se obtendrían más eficientemente los resultados del proyecto.

Se trabajó durante poco más de un mes sobre este simulador, estudiando sus módulos y modificando el código en puntos clave para obtener nuevas estadísticas. Se logró obtener nuevos datos como salida del simulador, que cuantifican los fallos producidos por *false sharing* [4].

Pasado poco más de un mes de trabajo, el dominio del simulador no era demasiado. Todo aparentaba a que, proyectando el mismo ritmo de trabajo durante el resto del plazo del proyecto, no se obtendrían resultados interesantes. Esto se debió principalmente a la complejidad del código, tanto algorítmica, como de estilo, debido a que actualmente hay múltiples colaboradores en el desarrollo del simulador. Además, durante este período de trabajo se estudiaron diferentes posibilidades para implementar el algoritmo principal del simulador, encontrando que modificar SESC para que utilice aquel que se encontró como óptimo, conllevara una cantidad de trabajo demasiado grande. Debido a este lento avance, se decidió realizar el análisis del esfuerzo necesario para desarrollar un nuevo simulador.

3.4. Cronograma de trabajo

En esta sección se presenta un cronograma aproximada del trabajo de esta parte del proyecto, donde se muestra el esfuerzo dedicado a cada tarea:

1. Agosto 2013:

    - Estudio de funcionamiento de procesadores multicore: Dado que este tipo de procesadores no son estudiados durante la carrera de grado, en los primeros

[4] En sistemas multiprocesador con memoria cache y memoria compartida, se denomina *false sharing* a la siguiente situación: si dos procesadores quieren hacer escrituras a bytes diferentes de un mismo bloque, cada escritura provocará una sobrecarga de comunicación, debido a que cada escritura exige la exclusividad en el uso del bloque. Se denomina *false sharing*, porque en realidad los procesadores no están compartiendo memoria, y por tanto el esfuerzo agregado no sería estrictamente necesario. Este fenómeno es responsable de una cantidad no despreciable de tráfico en estos procesadores [1], razón por la cual su cuantificación es de interés.

meses del proyecto se estudió su funcionamiento, la organización de los mismos, cómo se utilizan desde el punto de vista del sistema operativo, cómo es el proceso de booteo, etc. El estudio se basó fundamentalmente en la lectura del libro 'Computer Architecture: A Quantitative Approach' de Hennessy y Patterson [1] y el manual de la arquitectura IA32 [42].

- Estudio del simulador SimpleScalar: Para poder realizar el estudio del estado del arte, primero se debía tener un mínimo de experiencia con simuladores de procesadores para poder evaluar los demás de forma correcta. Se estudió el funcionamiento del simulador SimpleScalar, pues es un simulador simple y no demasiado abarcativo. Se estudió el código del mismo, lo cual permitió aprender no sólo de simuladores, sino de organizaciones avanzadas de procesadores superescalares.

2. Setiembre 2013:

   - Estudio de redes NoC: el estudio de los multiprocesadores mostró que para lograr su comprensión, es necesario tratar el tema de redes de interconexión en el propio chip, o sea las NoC (Network on Chip). El estudio de estas redes se basó en la lectura del libro 'Principles and Practices of Interconnection Networks', de Dally y Towles [5], el cual es sumamente citado en los artículos científicos del área.
   - Estudio del estado del arte en simuladores de procesadores: Se estudiaron los artículos de presentación de simuladores en las principales conferencias internacionales [2], [3] [6] [7] [8].

3. Octubre 2013:

   - Continuación de estudio de estado del arte en simuladores de procesadores: En esta etapa se leyeron artículos de diseño de procesadores y microarquitectura, con dos objetivos: primero, entender cuáles eran las principales líneas de trabajo en el área a nivel internacional; segundo, ver qué simuladores estaban siendo mayormente usados para validar las propuestas.
   - Estudio de simuladores SESC y ESESC: Se instalaron los simuladores SESC y ESESC y se estudió su código para analizar la factibilidad de su extensión y ganar conocimiento sobre ellos.

4. Noviembre 2013:

   - Estudio de simuladores gem5 y Graphite: Se instalaron los simuladores gem5 y Graphite para analizar la factibilidad de su extensión y ganar conocimiento sobre ellos.

- Desarrollo con SESC: Dado que se eligió el simulador SESC para como simulador a extender, se trabajó en él, estudiando su código y modificándolo para incorporar nuevas funcionalidades, las cuales fueron mencionadas en puntos anteriores de este documento.

5. Diciembre 2013:

   - Continuación de desarrollo con SESC.
   - Análisis de requerimientos para desarrollo de SimCo: El trabajo realizado en esta etapa está documentado en la siguiente sección.

4. Desarrollo de SimCo

En esta sección se realizará la descripción del proceso de desarrollo de SimCo.

4.1. Requerimientos

La aplicación deberá implementar un caso de uso fundamental, que es la correcta simulación de un programa escrito en un cierto lenguaje ensamblador, dada la especificación de un sistema computacional. Para ejecutar la aplicación, se deberá dar la configuración del sistema a simular mediante un archivo de configuración, de formato propio pero similar a un archivo de extensión properties, y brindar un programa en el lenguaje ensamblador especificado en la configuración. La aplicación proveerá datos de la simulación, como tiempo de ejecución, cantidad de accesos a memoria, etc. También proveerá archivos de salida con otras estadísticas, así como una traza de la simulación. Dado que es deseable que el simulador pueda ser utilizado con fines educativos, se hará énfasis en la sencillez de la configuración y en la presentación de los resultados.

Teniendo la anterior descripción como punto de partida se pueden detallar los siguientes requerimientos.

4.1.1. Requerimientos funcionales

1. Configurabilidad de la arquitectura: El simulador debe permitir especificar en su archivo de configuración con qué arquitectura se trabajará.

2. Configurabilidad de los núcleos de ejecución: Se podrá establecer cuántos núcleos de ejecución se simularán, indicando para cada uno cuál será la implementación que se utilizará. Según el tipo podrán variar las variables a parametrizar, pero se espera que como mínimo, para un procesador se especifique:

    - Nombre del procesador
    - Nombre del bus o red por el cual se accede a la memoria de instrucciones
    - Nombre del bus o red por el cual se accede a la memoria de datos
    - Valor inicial del program counter
    - Cantidad de ALUs de números enteros disponibles y su latencia
    - Cantidad de ALUs de punto flotante disponibles y su latencia
    - Cantidad de multiplicadores/divisores enteros y su latencia
    - Cantidad de multiplicadores/divisores de punto flotante y su latencia

3. Configurabilidad del sistema de memoria: Se permitirá configurar un sistema de memoria consistente en varios niveles de cache y memoria principal. Se deberá poder especificar si se trabajará con memoria centralizada o distribuida, debiéndose indicar en este último caso qué rangos de direcciones se asocian a cada dispositivo de memoria DRAM. En cualquiera de los dos casos, para cada DRAM configurada se deberá especificar como mínimo:

   - Nombre del dispositivo
   - Número de puertos de lectura/escritura
   - Latencia en ciclos
   - Capacidad en bytes (potencia de dos)
   - Nombre del bus o red que lo conecta con el resto del sistema

   Por otro lado, para cada cache se podrá especificar:

   - Nombre del cache
   - Número de puertos de lectura/escritura
   - Latencia en ciclos
   - Cantidad de conjuntos del cache (potencia de dos)
   - Asociatividad de cada conjunto
   - Tamaño de la línea en bytes (potencia de dos)
   - Política de reemplazo a utilizar (random, FIFO, LRU)
   - Política de escritura (writeback, writethrough)
   - Protocolo de coherencia de cache a implementar (MSI, MESI, Directorio, etc)
   - Nombre del bus o red que lo conecta con el elemento superior de la jerarquía de memoria
   - Nombre del bus o red que lo conecta con el elemento inferior de la jerarquía de memoria

4. Configurabilidad de la interconexión de procesadores y memoria: Se deberá poder configurar los medios compartidos a utilizar entre los diferentes elementos de memoria y los procesadores, de forma tal de poder especificar organizaciones tales como las de las figuras 2.4 y 2.5. Para los buses, si bien los parámetros a configurar variarán según qué implementación se elija (buses maestro esclavo, buses en pipeline, buses split-transaction, etc [5]), se mantendrán algúnos parámetros comunes a configurar, dentro de los cuales, los más elementales son:

   - Nombre del bus
   - Tipo de bus utilizado
   - Ancho del bus en bytes

- Algoritmo de arbitraje a utilizar

Para las redes de interconexión basadas en switches se deberán especificar:

- Algoritmo de enrutamiento a utilizar y las direcciones de cada uno de los elementos de la red.
- Especificación de la topología de la red, indicando o bien la disposición individual de cada switch, o mediante una descripción global (grid, torus, butterfly, etc).
- Especificación de los switches a utilizar.

5. Generación de trazas: El simulador deberá generar como salida una traza que describa el estado del sistema computacional en todo momento.

6. Generación de archivo de estadísticas: El simulador deberá generar como salida un archivo con información estadística de la ejecución, como tiempo total de ejecución, cantidad de instrucciones ejecutadas, cantidad de accesos a memoria, etc.

7. Visor de trazas: Una vez terminada la simulación, se deberá poder visualizar la traza generada a través de una interfaz gráfica, de forma que permita identificar fácilmente el estado de cada componente en todo momento.

4.1.2. Requerimientos no funcionales

1. Modularización: La aplicación deberá tener un diseño modular que permita el desarrollo e intercambio de componentes, los cuales deberán implementar una interfaz común. Ejemplos de estos componentes pueden ser nuevas arquitecturas, procesadores, dispositivos de memoria, tipos de redes de interconexión, etc.

2. Velocidad de ejecución: Si bien para los primeros usos el tiempo de ejecución del simulador no será crítico, es probable que eventualmente sea utilizado con fines de investigación, donde la velocidad de ejecución sí es crítica para una ágil exploración del espacio de diseño. Por esta razón se tomarán donde se pueda decisiones que mejoren la velocidad de ejecución de la aplicación.

3. Bajo número de dependencias: para asegurar la portabilidad y facilitar la instalación en el mayor número de ambientes posible, la aplicación deberá disminuir donde sea posible las dependencias hacia bibliotecas no estándares.

4.1.3. Metodología y condiciones de implementación

La aplicación se desarrollará siguiendo el paradigma de orientación a objetos, pues la estructura de herencia permite satisfacer de forma sencilla el requerimiento de modularidad.

El simulador se desarrollará en C++ por ser un lenguaje de programación compilado (rápida ejecución) y con soporte de orientación a objetos. El visor de trazas será implementado en Java, utilizando el paquete *Swing* para el desarrollo de la interfaz gráfica.

La aplicación se desarrollará siguiendo la filosofía de código abierto y estará disponible en GitHub en su repositorio [23]. Además, el código se escribirá con un esquema de nombres de variables y funciones en inglés para facilitar la comprensión del código por cualquiera que desee extenderlo.

4.2. Análisis y Diseño

En la siguiente sección se mencionarán aspectos de análisis y diseño de SimCo.

4.2.1. Diagrama de clases

En esta subsección se detalla el resultado del análisis del dominio a implementar, así como algunos diagramas que ilustran las relaciones entre las principales clases. Dado el gran número de clases que forman parte del modelo de dominio, el mismo se presentará de forma dividida. La arquitectura de la aplicación se puede descomponer en algunas clases principales:

- ISA: Clase abstracta, base de cualquier set de instrucciones. Provee funciones para traducir instrucciones de binario y String al modelo genérico de instrucciones del simulador y viceversa. Además provee funciones de construcción del set de registros de la arquitectura.
- ComputationalSystem: Clase principal de la aplicación, contiene referencias hacia todos los elementos relevantes de la simulación.
- MemorySystem: Esta clase mantiene la colección de los diferentes dispositivos de memoria que integran el sistema, así como un mapa de memoria (para los casos de sistemas de memoria distribuida), el cual indica qué direcciones globales se mapean a qué dispositivo.
- MemoryDevice: clase base para todos los dispositivos de memoria (DRAM, cache), contiene algunas variables comunes a estos, como latencia, número de puertos, y

mantiene estadísticas como número de accesos, cantidad de lecturas, de escrituras, etc.

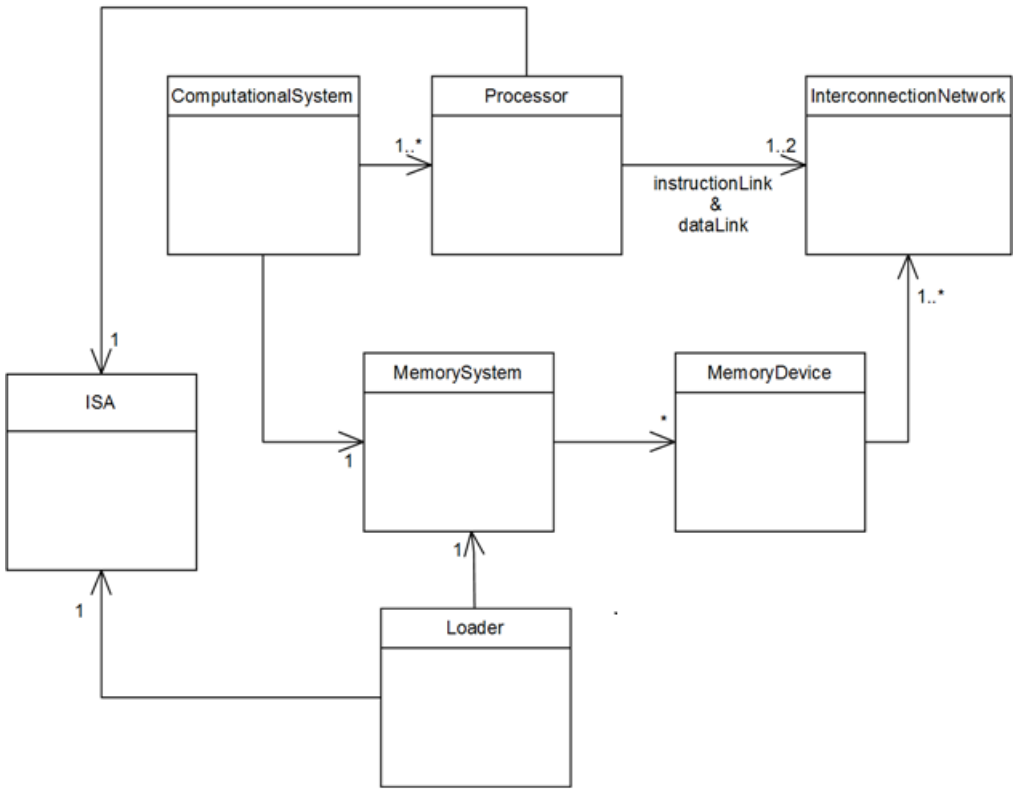


Figura 4.1: Diagrama de clases - Componentes principales de la aplicación

- Processor: Clase base para todos los procesadores a soportar. Contiene algunos elementos que se prevee serán comunes a todos los procesadores: un program counter, la referencia al set de registros de la arquitectura y links hacia los objetos de tipo 'InterconnectionNetwork' a través de los cuales se accede a la memoria de datos e instrucciones.
- Interconnection Network: Clase abstracta, base de todas las interconexiones entre dispositivos del sistema. Las implementaciones más usuales de este clase son Bus y P2PLink. Un bus conecta N dispositivos entre si, mientras que un P2PLink conecta dos switches, un switch y un dispositivo de memoria, o un switch y un procesador.
- Loader: Clase abstracta, base para cualquier loader, tiene la responsabilidad de leer el archivo con el programa a ejecutar y colocarlo en memoria.
- ConfigManager: Clase Singleton encargada de realizar el parseo y carga de la configuración.

En la figura 4.1 se presentan el diagrama que muestra la interacción de los componentes más importantes de la aplicación.

Modelo genérico de arquitectura

Lograr que la arquitectura sea un parámetro configurable es particularmente complejo (y una de las razones por las cuales la gran mayoría de los simuladores disponibles en el ámbito académico soporta sólo un set de instrucciones) puesto que en la realidad, una microarquitectura es la implementación de una arquitectura y no al revés. Esto en sentido estricto parece sugerir que se debe implementar un procesador diferente para cada set de instrucciones soportado. Afortunadamente esto no es del todo obligatorio, ya que muchos sets de instrucciones poseen características en común (tipos de instrucciones, operandos, etc), en particular aquellos de arquitecturas RISC. De este modo, es posible desacoplar el set de instrucciones de su implementación, modelando las instrucciones de forma genérica, como se muestra en la figura 4.2. Será responsabilidad del diseñador encargado de extender el simulador con una nueva arquitectura el decidir si utilizará una implementación de un procesador ya existente, o si implementará una nueva.

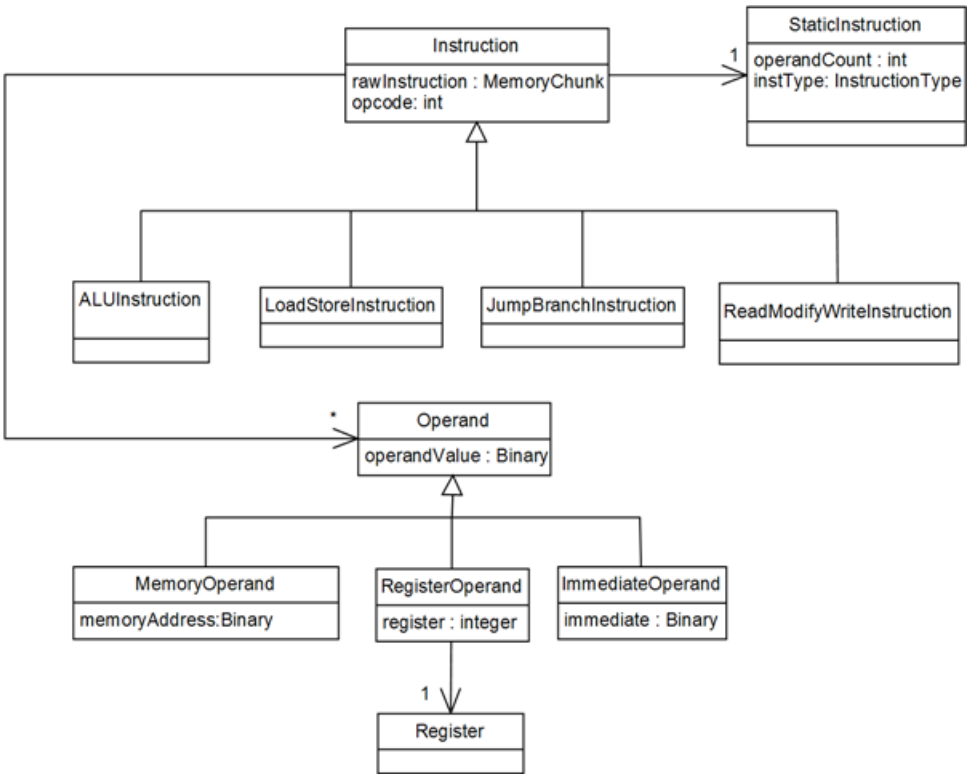


Figura 4.2: Diagrama de clases - Modelo genérico de instrucción

Genéricamente, una instrucción está formada por un código de operación (opcode) y un conjunto de argumentos (operandos). Modelando los operandos de forma genérica y con una jerarquía de instrucciones, se puede implementar la mayoría de las instrucciones de muchas arquitecturas al mismo tiempo.

Con esta solución, para aceptar un nuevo set de instrucciones, se debe implementar una subclase de 'ISA' que provea los métodos de conversión de instrucciones de binario a la estructura de clases presentada anteriormente, una clase que herede de Loader que realice la carga desde un archivo de texto, y la ejecución de las instrucciones únicas de la arquitectura, dentro del procesador.

Es digno de mencionar que el modelo presentado no contempla todos los sets de instrucciones existentes. En particular no permite modelar aquellos de las arquitecturas EPIC [21] aunque puede ser extendido para hacerlo (por ejemplo, definiendo una clase hija de instrucción la cual se asociará con un conjunto de instrucciones).

Para esta versión de SimCo se implementó una versión reducida de la arquitectura MIPS32. La especificación de esta arquitectura se encuentra en [30]. Se decidió incorporar esta arquitectura por dos razones: es una arquitectura RISC, muy simple, y es utilizada en algunos cursos del Instituto de Computación [31]. Dado lo extenso del proyecto, es importante no agregar complejidad en áreas no medulares, por lo tanto la elección de una arquitectura simple es imperativa. La especificación de la arquitectura se implementa en dos clases: MIPS32ISA, la cual extiende la clase ISA y contiene la especificación del set de instrucciones, y MIPS32Loader, la cual extiende la clase Loader y es la encargada de realizar la carga de las instrucciones desde archivo a la memoria del simulador, implementando funciones de ensamblador, como resolver etiquetas. Utiliza las funciones de MIPS32ISA para realizar las traducciones a binario.

Modelo de red de interconexión

Con respecto a la red de interconexión, el diseño se enfocará en poder respetar el principio de modularidad, de forma tal de que un CPU o memoria se pueda conectar al medio de comunicación a través de una interfaz común, sin importar si éste está implementado por un bus (de cualquier tipo) o por un enlace punto a punto que transmite paquetes. Lograr tal interfaz para todas las posibles interacciones es al menos intrincado, ya que se debe rescatar qué comparten una red de pasaje de mensajes y un bus con arbitraje centralizado con líneas de direcciones y de datos. La estrategia utilizada fue modelar cualquier transferencia en algún medio como un *mensaje*, pudiendo definir de este modo un método *submitMessage* que tome un mensaje cualquiera.

Para modelar también el arbitraje del medio compartido, se definió un método *requestAccess*, el cual es utilizado como punto de sincronización para la función de arbitraje del medio compartido. Esta decisión no afecta la implementación de buses más simples donde tal función no aplique, pudiendo la implementación simplemente llamar al método *accessGranted* del solicitante. Para soportar la existencia de dicho método sea cual sea el dispositivo que solicita acceso al medio, se define una interfaz *IMessageDispatcher* la cual implementarán todas las clases que tengan que hacer uso de algún medio compartido (procesadores, memorias, controladores de diccionario, switches, etc).

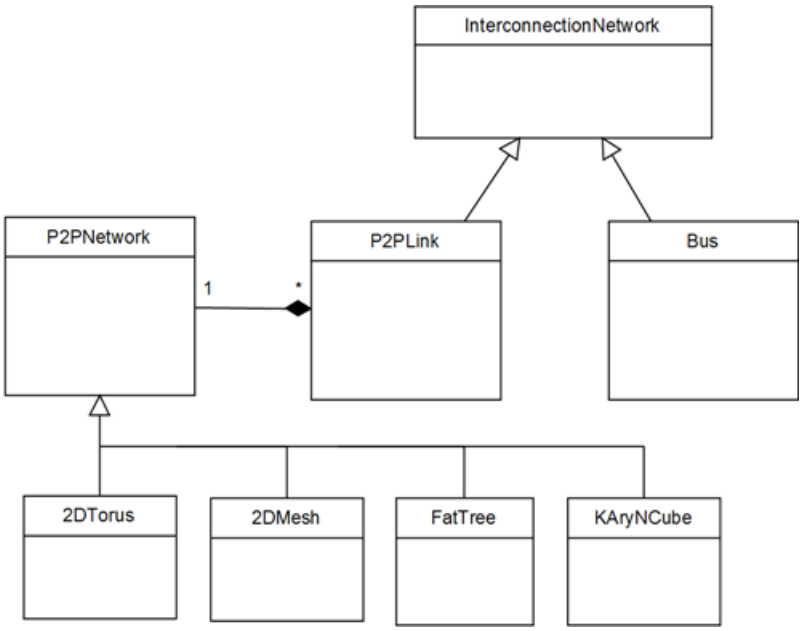


Figura 4.3: Diagrama de clases - Red de interconexión

Por último, para el caso de redes basadas en conmutadores (switches), se define una clase *P2PNetwork* que tendrá la información de la topología de la red implementada y será consultada por los conmutadores al momento de realizar los reenvíos. Además detectará la aparición de deadlocks o livelocks en configuraciones que lo permitan.

4.2.2. Algoritmo de simulación

Para la elaboración del algoritmo de simulación se tomaron en cuenta algunas sugerencias y algoritmos presentados en [22], en conjunción con ideas utilizadas en otros simuladores como el SimpleScalar y SESC.

La unidad de tiempo a utilizar principalmente en el simulador es el ciclo de reloj, es decir que el loop principal de la aplicación tiene el siguiente pseudocódigo:

while no termine la simulación **do**
    simular ciclo de reloj
end while

Donde la condición de finalización de simulación está dada o bien porque se llegó al límite de ciclos ingresado en la ejecución, o porque todos los procesadores culminaron la ejecución de sus respectivas porciones de código.

Si bien el algoritmo principal parece trivial, la complejidad se presenta al respecto del refina-

miento de dicho pseudocódigo. Dado que en cada ciclo de reloj se simularán varias acciones que ocurren de forma concurrente y de las cuales varias son mutuamente dependientes, se deberá mantener durante toda la iteración el estado del sistema al inicio del ciclo de reloj y además los cambios calculados para el próximo ciclo (si se utilizara una única variable se podrían calcular resultados en función de valores que no deberían ser 'visibles' aún para los eventos del simulador). Por esta razón, el pseudocódigo de cada iteración será en principio el siguiente:

for cada entidad del simulador **do**
    actualizar inicio de ciclo
end for
while queden eventos para simular **do**
    simular evento
end while

Donde 'actualizar inicio de ciclo' incluye, como fue mencionado anteriormente, actualizar el estado de cada entidad simulada con los cambios calculados durante el ciclo anterior. El *while* siguiente en el pseudocódigo se puede interpretar simplemente como realizar las acciones del sistema computacional para ese ciclo. Se decidió utilizar eventos como unidad de simulación en lugar de iterar sobre las entidades simuladas y realizar las acciones correspondientes, fundamentalmente para desacoplar el orden en que deben ser ejecutados los eventos de los eventos en si. De este modo, cuando se simula una acción en un componente que desencadena una en otra entidad, simplemente dispara un evento y lo agenda en el simulador en el tiempo correspondiente. Por ejemplo, cuando un dispositivo de memoria recibe un pedido de una palabra de memoria, si la latencia de este fuera nula, la devolución podría ser inmediata mediante una llamada a la función *submitMessage* del medio compartido que la conecta al resto del sistema y dicha llamada podría estar escrita directamente en el código, sin embargo, si la latencia fuera t mayor que cero, esa llamada tendría que realizarse t ciclos después y por tanto la llamada no podría estar codificada del mismo modo. Con el sistema de eventos, ambas pueden ser tratadas de la misma manera, simplemente cambiando la cantidad de ciclos en la cual se debe realizar la llamada.

4.3. Análisis de riesgos y Plan de desarrollo

La aplicación se desarrollará siguiendo una metodología iterativa e incremental. En cada iteración se propondrá el desarrollo de algún módulo del sistema, pues dado que existen pocos casos de uso en los requerimientos y que todos utilizan el sistema de modo integral, no es práctico realizar la división por casos de uso.

Durante la formulación de los requerimientos, la preocupación fundamental que surgió fue el no saber si el tiempo disponible era suficiente para realizar el desarrollo. Para mitigar ese riesgo, se planificó un cronograma exigente, en el cual se organizó el desarrollo de las

características más particulares de SimCo en primer lugar (el modelo genérico de arquitectura y la interconexión), y la implementación de procesadores luego. De esta forma, si el alcance resultara excesivo, se podría limitar sobre la marcha y aún contar con un producto útil (que por lo menos permitiera estudiar el desempeño de un sistema de memoria en multiprocesadores).

Cronograma de desarrollo

A continuación se muestra el cronograma de desarrollo como fue implementado SimCo. Al momento de la planificación se definió como objetivo realizar iteraciones de dos semanas. Sabiendo esto y observando las duraciones de las iteraciones, se puede obtener la desviación de cada una según lo implementado.

1. Iteración 0: 16/01/2014 - 10/02/2014

    - Desarrollo de módulo de configuración.
    - Implementación del set de instrucciones MIPS32, así como su parser y ensamblador.

2. Iteración 1: 10/02/2014 - 24/02/2014

    - Interfaz de eventos propia del simulador
    - Algoritmo principal del simulador
    - Interfaces de redes de interconexión y dispositivos de memoria.
    - Implementaciones de bus y memoria RAM como implementación de interfaces anteriores.

3. Iteración 2: 24/02/2014 - 17/03/2014

    - Arbitraje de buses.
    - Implementación de memorias cache
    - Implementación de un protocolo de coherencia de cache para memoria de acceso uniforme, por ejemplo bus snooping.

4. Iteración 3: 17/03/2014 - 07/04/2014

    - Switches.
    - Algoritmo de enrutamiento 'Direction Order Routing' para topología de malla de dos dimensiones.
    - Controlador de 'Directorio' para protocolo de coherencia de cache basado en directorios.

5. Iteración 4: 07/04/2014 - 28/04/2014

    - Implementación de CPU simple
    - Pruebas del sistema con el CPU integrado

6. Iteración 5: 28/04/2014 - 09/06/2014

    - Carga desde archivo de todos los componentes del simulador
    - Desarrollo del visor de trazas
    - Pruebas completas con varios procesadores y jerarquía de memoria multinivel.

Desviaciones y estado de la aplicación

Dado que cada iteración se planificó para ser implementada en dos semanas, del cronograma final de implementación se puede deducir la desviación obtenida durante cada etapa.

El desarrollo de los módulos propuestos para cada etapa se encuentra completo, a excepción del módulo de redes basadas en switches, el cual si bien es utilizable en el simulador, su verificación no fue completa y hay algunas fallas documentadas en el código. A su vez, la visión de los resultados de estas redes en la aplicación SimcoViewer aún no está completa.

4.4. Consideraciones de implementación

En esta sección se incluirán algunos aspectos de la implementación dignos de comentar, algunos por ser vitales para el uso del simulador y otros por haber exigido decisiones interesantes a nivel de programación.

4.4.1. Implementación del sistema de eventos

Dado que el disparo de un evento consiste en agendar una función a ser ejecutada durante un cierto tiempo posterior, una implementación sencilla del sistema de eventos sería poder guardar funciones como si fuera cualquier otro objeto en alguna estructura de tipo calendario. Como C++ no permite el tratamiento de funciones como parámetro a otras funciones (como se realizaría en algún lenguaje de programación funcional como Haskell [25]), la estrategia abordada fue definir una interfaz *IEventCallback* con una única función *simulate* y una clase que herede de ella por cada tipo de evento existente en el sistema, que en la implementación de *simulate* llame a la función correspondiente. De este modo, en

el bucle principal del simulador se puede mantener una estructura de objetos *IEventCallback*, los cuales al invocárseles el método *simulate*, gracias al polimorfismo de esta función, provocarán la invocación a la función correspondiente.

Como para cada ciclo puede existir un número no acotado de eventos, la estructura elegida para mantener los eventos para cada ciclo es una cola. A su vez, se guarda una cola de estos eventos para cada ciclo en los que existan eventos agendados.

4.4.2. Sistema de memoria

Al momento de simular un procesador con una cantidad cualquiera de memoria, aparece la dificultad de que quizás la máquina que corre la simulación disponga de menos memoria de la que se desea simular (por ejemplo al simular sistemas de memoria distribuida). Dada esta realidad, la estrategia será mantener en memoria del simulador sólo aquellas posiciones de memoria de la máquina simulada que han sido escritas en algún momento. Para lograr este objetivo, los dispositivos de memoria DRAM simulados se implementaron con una tabla multinivel de memoria, como se muestra en la figura 4.4.

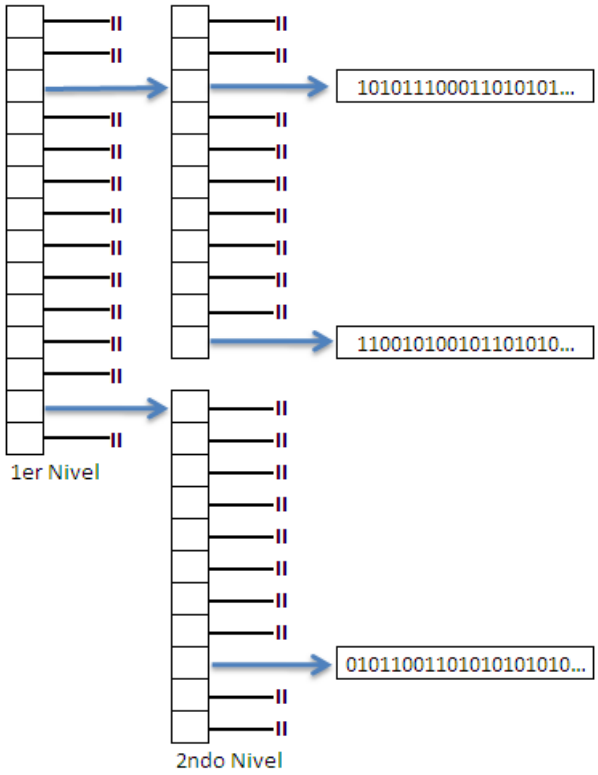


Figura 4.4: Tabla de memoria multinivel

Esta estrategia, utilizada extensamente en sistemas de memoria virtual para mantener las

tablas de páginas, permite reducir drásticamente la memoria utilizada en caso de que sólo un subconjunto de la memoria esté en uso, y aún así mantener de forma ordenada el espacio de direccionamiento (es decir que es fácil obtener el valor de una palabra y las siguientes dada una dirección) sin el overhead de otras estructuras de datos como un diccionario hash.

4.4.3. ARQUITECTURA MIPS32

Como fue especificado anteriormente, en esta primera versión de SimCo se implementó la arquitectura MIPS32 [30]. La incorporación de dicha arquitectura exigió la implementación de varias clases del modelo genérico de instrucción de la figura 4.2. Se implementaron las siguientes clases hijas de Instruction:

- ALUInstruction: Esta clase representa una instrucción de ALU. Contiene un arreglo de operandos origen y un arreglo de operandos destino, permitiendo de este modo modelar con la misma clase instrucciones de arquitecturas de dos o tres vías. Además se permiten modelar instrucciones que afecten a varios registros (por ejemplo de arquitecturas vectoriales). Del set de instrucciones implementado, se mapean a esta clase las siguientes instrucciones: add, addu, addi, addiu, and, andi, nor, or, ori, sll, sllv, sra, srav, srl, srlv, sub, subu, xor, xori, lhi, llo, slt, sltu, slti, sltiu, mfhi, mflo, mthi, mtlo, nop. Como los operandos son genéricos, con esta clase se pueden implementar instrucciones que incluyan operandos en memoria.
- LoadStoreInstruction: Esta clase implementa una instrucción de acceso a memoria en arquitecturas Load/Store [5]. Si bien el efecto deseado puede ser logrado con una instrucción de ALU que no realice ningún efecto, se decidió implementar una clase nueva pues típicamente estas instrucciones utilizan unidades funcionales diferentes que las instrucciones de tipo ALU. Del set de instrucciones implementado se mapean a esta clase las siguientes instrucciones: lb, lbu, lh, lhu, lw, sb, sh, sw.
- BranchInstruction: Esta clase permite modelar una instrucción de salto condicional. Incluye un operando que indica la dirección destino, uno o más operandos que implican la condición a evaluar y por último un parámetro de tipo enumerado que indica qué tipo de condición se debe chequear para saber si el salto es tomado o no. Del set de instrucciones implementado se mapean a esta clase las siguientes instrucciones: beq, bgtz, blez, bne.
- JumpInstruction: Esta clase implementa una instrucción de salto incondicional. Se decidió implementar una clase separada de las de salto condicional, debido a que su tratamiento por procesadores supererscalares es diferente. Por ejemplo, las instrucciones de salto incondicional marcan un cambio de flujo en la etapa de decodificación,

[5] Las arquitecturas Load/Store son aquellas que acceden al sistema de memoria únicamente a través de las instrucciones clásicas Load (cargar desde memoria) y Store (guardar hacia memoria). MIPS32 es una arquitectura Load/Store.

mientras que en los saltos condicionales el resultado se conoce recién luego de la ejecución. Del set de instrucciones implementado se mapean a esta clase las siguientes instrucciones: j, jal.

Además de las instrucciones, se debieron incorporar cuatro subclases de operando:

- RegisterOperand: Esta clase representa un operando directo a registro. Contiene el índice del registro operando y un puntero a dicho registro (el cual es cargado durante la etapa de decodificación).
- ImmediateOperand: operando inmediato. Contiene el valor del operando.
- IndexedOperand: Esta clase implementa un operando en memoria, cuya dirección se forma por un registro más un desplazamiento. Contiene los mismos datos que un 'RegisterOperand' y un 'ImmediateOperand'.
- SpecialRegisterOperand: Esta clase es utilizada para representar operandos de registro particulares de la arquitectura. En el caso de MIPS32 se utiliza para instrucciones que acceden al registro PC, LO o HI [6].

4.4.4. Ancho de la arquitectura simulada

El ancho de una arquitectura especifica entre otras cosas, de cuántos bits disponen sus registros. De este modo, si una arquitectura es de 64 bits, sus registros serán típicamente de 64 bits. Actualmente la mayoría de las arquitecturas son de 32 y 64 bits y a lo largo de la historia se han construido también máquinas de 16 y 8 bits. Esto indica que eventualmente podrían existir máquinas de 128 bits o más grandes aún [7], por lo tanto sería interesante que SimCo aceptara tales características. Dicha incorporación no es trivial: si se considera la jerarquía de instrucciones presentada en 4.2, el valor del operando 'Inmediato' diferirá en tamaño según el ancho de la arquitectura (pudiendo requerir hasta 128 bits en arquitecturas de este tamaño). A su vez, la variable con la cual se modelará el valor de cada registro también puede exigir 128 o más bits de memoria en arquitecturas de ese ancho. Para la elección del tipo de dato con el cual se guardarán esos valores se consideraron los siguientes candidatos:

1. Un arreglo de tamaño variable determinado durante la lectura del archivo de configuración (implementable como un char*). Este tipo de dato tiene la ventaja de que permitiría modelar cualquier tamaño de arquitectura, aunque tiene la desventaja de presentar una complejidad algorítmica importante (se deben redefinir todos los operadores), lo cual dificulta su implementación y penaliza el tiempo de ejecución.

[6] LO y HI son registros de 32 bits donde se colocan los resultados de una multiplicación o división. En LO se ubican los 32 bits menos significativos del resultado y en HI los 32 más altos.

[7] Actualmente, algunas máquinas vectoriales disponen de algunos registros de 128 bits. A su vez, el sistema AS/400 [32] maneja punteros como datos de 128 bits

2. Una variable del tipo 'long int' de C++, la cual es guardada en memoria con 64 bits en la mayoría de los sistemas. Ésta solución *no* permite la incorporación de arquitecturas de 128 y más bits pues dicha variable no es suficiente para guardar el valor completo de un registro. Sin embargo, presenta la enorme ventaja de ser de muy simple y de rápida manipulación (es posible usar los operadores definidos en el lenguaje C++ para operarlas).

Dado el análisis anterior se decidió implementar los anchos de los registros y demás valores críticos con variables 'long int', fundamentalmente para no agregar más complejidad aún a la implementación. No se descarta un cambio en la implementación en algún futuro y por lo tanto se deja presentada una alternativa que aborde el problema de forma general.

4.4.5. Memoria Cache

La implementación de la memoria cache fue de los módulos algorítmicamente más complejos de realizar. Para representar las líneas del cache se utilizó una tabla donde cada entrada representa una línea del cache, la cual contiene la siguiente información:

- Tamaño de la línea en bytes (int)
- Etiqueta que identifica el bloque ubicado en dicha línea (long)
- Datos contenidos en la línea (char*)
- Timestamp de último uso (unsigned long)
- Bandera indicando si el valor contenido fue modificado (boolean)
- Bandera indicando que la línea está por ser reemplazada (boolean)
- Estado de la línea según el protocolo de coherencia usado (enumerado)

Al recibir un pedido de lectura o escritura, la primer acción a realizar es obtener el valor de la etiqueta, conjunto y desplazamiento de la dirección. Luego se revisan las líneas del conjunto correspondiente para buscar si existe una línea con misma etiqueta que la de la dirección buscada. Si existe coincidencia se produce un *hit* y se proceden a realizar las acciones correspondientes: si se trata de una lectura, se obtienen los valores a leer de la línea y se envían por la conexión superior. En la implementación actual, se controla que el acceso esté confinado a una única línea, lanzándose una excepción de acceso no alineado en caso de que la lectura abarque varias líneas. Ésta restricción es normal para las arquitecturas MIPS, sin embargo, provocará problemas para arquitecturas más libres como x86, quedando como trabajo futuro el abordaje al problema [8]. Si se trata de una escritura, se actualiza el valor de la línea con los datos enviados en el pedido de escritura. En caso de utilizarse la política *writeback*, se marca la línea, indicando que contiene valores que aún no han sido

[8] El simulador Zesto presenta una solución bien documentada al problema de accesos de memoria que involucren varias líneas de cache [34].

actualizados en los niveles inferiores de la jerarquía de memoria. Si se utiliza la política de escritura *writethrough*, se propaga la escritura hacia la siguiente memoria.

En caso de producirse un fallo en el acceso, la primera acción es ubicar una línea del cache para colocar el nuevo bloque de memoria a cargar. Si existe alguna línea libre en el conjunto correspondiente, se la asigna. De lo contrario se selecciona alguna linea a liberar mediante la política de reemplazo utilizada. En cualquier caso, la línea que está por ser reemplazada se la marca como *efímera*, posteriores lecturas y escrituras sobre líneas efímeras se tratan de forma normal, sin embargo, en el anexo se presenta una idea de diseño al respecto de las líneas efímeras que puede ser explorada con el simulador. Una vez elegida la línea que ubicará el bloque accedido, se inicia un pedido de lectura de bloque hacia el siguiente dispositivo de memoria. Cuando este es devuelto, se escribe en la línea seleccionada, realizando una escritura del bloque en caso de que este se encuentre modificado y se utilice la política de escritura *writeback*.

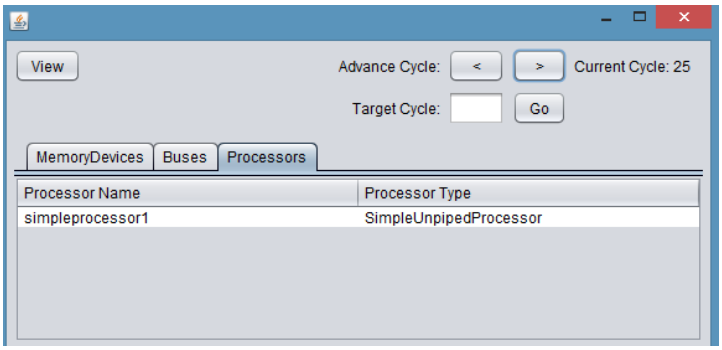
Todas las escrituras que se deben realizar sobre los niveles inferiores de la jerarquía se realizan a través de un buffer de escritura [9], el cual actualmente tiene largo infinito. Como trabajo futuro se debería permitir configurar la cantidad de mensajes permitidos en dicho buffer, así como definir qué hacer en caso de que se deba realizar una escritura en un nivel inferior de memoria y el buffer de escritura se encuentre completo.

4.4.6. Archivo de configuración

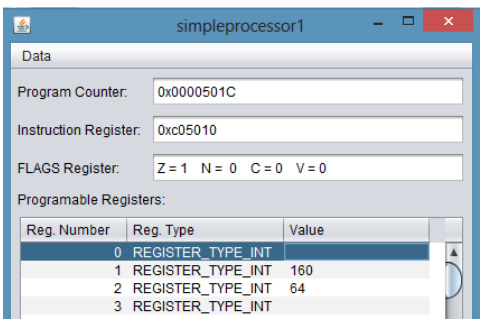
La sintaxis soportada en el archivo de configuración hasta el momento es sumamente simple. La misma consiste en un archivo de formato similar a *properties* [26], con pequeñas directivas agregadas: las líneas con una única palabra inician o cierran un bloque de definición de entidad de simulación. Por ejemplo para definir un cache se debe escribir una línea con la palabra reservada *cache* y en las líneas siguientes los parámetros configurables del cache, seguidas de una línea con la palabra *end*. Un ejemplo de archivo de configuración se puede encontrar en el anexo.

Dada la simpleza de la configuración también se encuentra el problema de que si se quiere definir un sistema con N procesadores, cada uno con dos niveles de cache privados, se debe escribir el mismo código N veces, cambiando los nombres por los correspondientes en cada caso. En un futuro se deberían incorporar formas más simples de realizar esta tarea, por

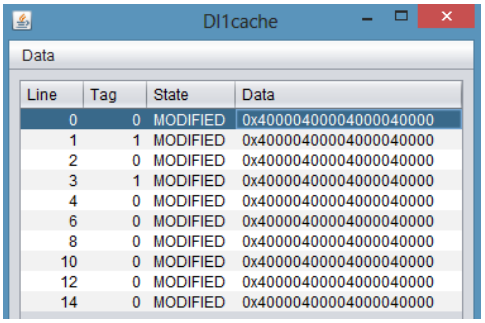
[9] Un buffer de escritura (*write buffer*) es una pequeña memoria incluida en los sistemas de memoria cache para almacenar temporalmente valores a escribir en niveles inferiores de la jerarquía. Su utilidad es no bloquear al CPU cuando algún acceso de este desencadena una escritura a un nivel inferior, dejando los valores a escribir accesibles en el buffer mientras son escritos en memoria principal. Cuando posteriormente se realice una lectura en cache y esta resulte en miss, se deberá chequear el buffer para verificar si no contiene el valor buscado, antes de buscar el dato en el nivel inferior de la jerarquía. Más sobre buffers de escritura se puede encontrar en [33]



(a) Principal



(b) Procesador



(c) Cache

Figura 4.5: Algunas ventanas de SimcoViewer

ejemplo, definiendo una sintaxis para arreglos de procesadores y memorias.

4.5. SIMCOVIEWER

SimcoViewer es el nombre del visor de trazas del simulador. El objetivo del mismo es presentar los resultados obtenidos a través de SimCo de un modo más amigable para el usuario, mediante una interfaz gráfica. Este punto, aunque menor para muchos programas, es en el proyecto de vital importancia, ya que una compleja presentación de los resultados puede provocar que el simulador no sea útil a nivel educativo.

Dado que los sistemas computacionales a simular pueden variar muchísimo, desde un microcontrolador con memoria principal únicamente, pasando por una organización multi-núcleo y varios sistemas de cache como en la figura 4.8, hasta clusters con una centena de núcleos de ejecución y una red de interconexión basada en switches, es claro que la interfaz gráfica deberá ser fundamentalmente flexible. Por esta razón se eligió que la pantalla principal del visor presente los diferentes elementos que componen el sistema computacional, pudiéndose abrir una nueva ventana para ver el estado ciclo a ciclo de cada uno. En la ventana principal existirán comandos que permitan avanzar el estado, el cual será reflejado en las diferentes entidades.

4.6. VERIFICACIÓN Y EJEMPLOS DE USO

El proceso de verificación es sumamente complejo para los simuladores de sistemas computacionales. Durante el estudio del estado del arte se comprobó que en general los simuladores no son validados contra hardware real, aunque esto puede llevar a errores importantes en la simulación, como muestran algunos estudios, por ejemplo [27]. Si bien se alega en algunos trabajos que la verificación contra hardware real no asegura la correcta simulación en todos los casos y que por tanto la dificultosa tarea no vale la pena, se entiende que dicha verificación es una importante fuente de eliminación de errores. Lamentablemente, dada la simpleza del procesador implementado, la comparación contra una máquina real no es posible aún, pues no es posible simular fielmente un procesador moderno. Por esta razón, la verificación realizada fue en base a la comparación de las trazas generadas por el programa contra sus respectivos valores esperados.

Los tests implementados se encuentran en el repositorio de SimCo [23], y consisten en:

- Tests unitarios para las estructuras de datos implementadas
- Test unitario de Cache, donde se verifican los valores esperados ante la lectura/escritura de diferentes posiciones de memoria.
- Pruebas unitarias para la arquitectura MIPS32 implementada, donde se verifica la correcta codificación/decodificación de las instrucciones, así como la correctitud del Loader encargado de parsear el archivo ensamblador y colocar dichar instrucciones en la memoria simulada.
- Pruebas de integración para el sistema de memoria, probando diferentes configuraciones de jerarquía de memoria (con uno, dos y tres niveles de memoria cache y diferentes políticas de escritura). Para aislar el comportamiento del sistema de memoria del resto, se utilizó una versión de debug de CPU (*DummyDispatcher*), la cual únicamente emite mensajes de lectura y escritura al sistema de memoria.
- Pruebas del sistema con un único procesador, donde se verifica la correcta ejecución del ciclo de instrucción para diferentes programas, así como la carga desde archivo de la configuración.
- Pruebas del sistema con múltiples procesadores, donde se verifican fundamentalmente los protocoos de coherencia de cache y las características de la implementación de bus que no se comprueban en los tests anterioers (como las funciones de arbitraje).

A continuación se muestran algunos ejemplos de uso de SimCo, los cuales servirán además como validación del funcionamiento del simulador, pues las salidas se verificarán contra resultados teóricos.

- Prueba funcional y visión del ciclo de instrucción

  En esta prueba se mostrará cómo SimCo puede ser utilizado para mostrar el resultado

de una ejecución. Esto podría ser útil en cursos de lenguaje ensamblador, para que los estudiantes pueden corroborar los resultados de sus programas. Para esta parte, el programa utilizado es el siguiente:

```
 ORG 0x5000
XOR R2, R2, R2 # se carga el registro R2 con valor 0
XOR R3, R3, R3 # se carga el registro R3 con valor 0
XOR R4, R4, R4 # se carga el registro R4 con valor 0
ADDI R2, R2, 1 # se suma 1 al registro R2
SLL R2, R2, 4 # se realiza un corrimiento de 4 lugares del registro R2
OR R3, R2, R3 # R3 = R2 OR R3 (equivalente a la asignación pues R3 tiene
el valor 0)
LLO R4, 1 # Se carga 1 en los bits menos significativos de R4
SUB R3, R3, R4 # R3 = R3 - R4
ADD R5, R3, R2 # R5 = R3 + R2
HALT
```

El código del programa es muy simple. En él, se realizan algunas operaciones aritméticas, las cuales están explicadas en los comentarios. Al simular la ejecución del programa en SimCo con cualquier configuración de memoria e interconexión y cargando la traza generada en la aplicación SimcoViewer, se pueden ver los resultados obtenidos por cada una de las operaciones. En la figura 4.6, se muestra una captura de pantalla al ejecutar la instrucción 'ADD', donde se pueden verificar los valores finales de los registros. Es sencillo observar que los resultados son los correctos.

- Prueba de resultados en cache:

  En esta prueba se mostrará la utilidad de SimCo para estudiar la utilización del sistema de memoria para un cierto programa. Se utilizará un uniprocesador con una jerarquía de memoria de tres niveles como el de la figura 4.8. A continuación se presenta la configuración de la jerarquía de memoria:

Cache L1 de instrucciones	
Sets:	8
Asociatividad:	2
Tamaño de línea:	16 bytes
Política de remplazo:	LRU

Cache L1 de datos	
Sets:	8
Asociatividad:	2
Tamaño de línea:	16 bytes
Política de remplazo:	LRU
Política de escritura:	writeback

Cache L2 Unificada	
Sets:	16
Asociatividad:	4
Tamaño de línea:	32 bytes
Política de remplazo:	LRU
Política de escritura:	writeback

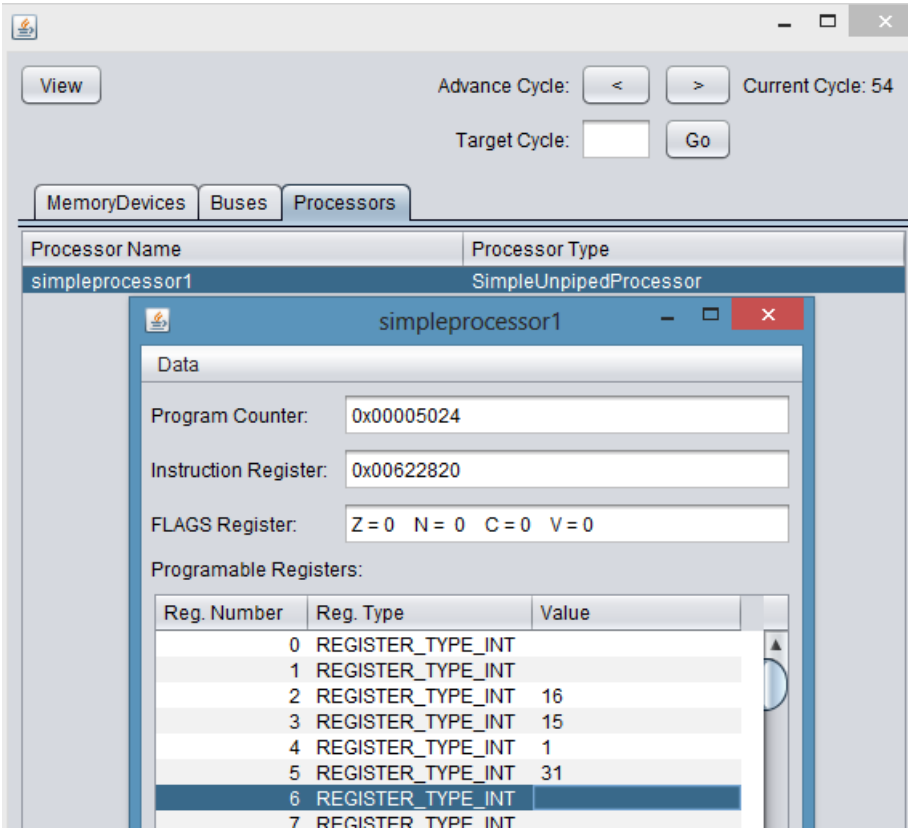


Figura 4.6: Estado final de registros en ejemplo 1

Cache L3 Unificada	
Sets:	32
Asociatividad:	8
Tamaño de línea:	128 bytes
Política de remplazo:	LRU
Política de escritura:	writeback

Como programa de prueba se utilizó el siguiente fragmento de código MIPS:

```
 ORG 0x5000
# Iniciación de variables
     ANDI R2, R2, 0
     LLO R2, 64
     ANDI R1, R1, 0
     LLO R1, 0x1000 ; dirección de comienzo 0x1000
# Itero palabras
loop:
     SW R2, 0, R1
```

```
    ADDI R1, R1, 4
    JAL loop
```

En dicho código, se realiza una iteración de la memoria a partir de la dirección 0x1000, escribiendo el valor 64 en cada palabra de memoria (cada palabra ocupa 4 bytes). Se configuraron los ciclos de simulación de forma tal que el loop se ejecute 250 veces. Al ejecutar, el simulador brinda los siguientes datos:

```
 Simple Unpiped Processor
Name: simpleprocessor1
Processor total executed instructions: 754
Processor total executed jump instructions: 250
Processor total executed alu instructions: 254
Processor total executed memory instrucitons: 250

Memory Cache
Name: L3cache
MemoryDevice Accesses: 33
MemoryDevice Reads: 33
MemoryDevice Writes: 0
Hit Count: 24
Miss Count: 9
Repl. Count: 0

Memory Cache
Name: L2cache
MemoryDevice Accesses: 112
MemoryDevice Reads: 65
MemoryDevice Writes: 47
Hit Count: 79
Miss Count: 33
Repl. Count: 0

Memory Cache
Name: Dl1cache
MemoryDevice Accesses: 250
MemoryDevice Reads: 0
MemoryDevice Writes: 250
Hit Count: 187
Miss Count: 63
Repl. Count: 47

Memory Cache
```

```
Name: Il1cache
MemoryDevice Accesses: 754
MemoryDevice Reads: 754
MemoryDevice Writes: 0
Hit Count: 752
Miss Count: 2
Repl. Count: 0
```

Análisis de resultados

Como indica la salida del simulador, se ejecutaron 754 instrucciones. Al analizar el flujo del programa, se puede ver que esto incluye las cuatro instrucciones de inicialización y 250 iteraciones del loop. Dado que se ejecutaron 754 instrucciones, se realizaron esa misma cantidad de lecturas sobre el cache de instrucciones. Dado que los bloques del cache IL1 tienen 16 bytes y que el programa comienza alineado con estos (la primer instrucción se encuentra al inicio de un bloque), todo el programa queda contenido en dos bloques de memoria, dando como resultado que de los 754 accesos, sólo dos resulten en *miss* (la primera vez que se requiere alguno de los dos bloques) y los restantes 752 sean *hits*.

Dado que el loop se ejecutó 250 veces, se realizaron 250 escrituras sobre el cache de datos de primer nivel, como se puede ver en los resultados. Como la memoria se recorre secuencialmente en el ejemplo, las direcciones accedidas son 0x1000, 0x1004, 0x1008, 0x100C, 0x1010, 0x1014, etc, las cuales están alineadas con los bloques (la dirección 0x1000 marca el inicio de un bloque pues es múltiplo de 16). Por cada cuatro escrituras sobre el cache, una será *miss* (la primer referencia a dicho bloque) y las siguientes resultarán en *hit*. Como cada cuatro accesos se accede a un nuevo bloque y

$$\#BloquesAccedidos = 250/4 = 62,5$$

, se accederán a 63 bloques de memoria, lo cual coincide con la cantidad de fallos indicados por la salida del simulador. Las restantes 187 escrituras resultarán en hits. El valor que resta por analizar es el del número de reemplazos: como el cache DL1 tiene 8 conjuntos, cada uno con 2 vías, en el cache se pueden colocar 16 bloques de memoria. Una vez lleno el cache, cada nuevo bloque a cargar implicará remover uno ya existente, por tanto habrán

$$\#BloquesAccedidos - \#BloquesEnCache = 63 - 16 = 47$$

reemplazos.

Para realizar el análisis sobre el cache de segundo nivel primero se contará la cantidad de accesos. Tanto los fallos en el cache L1 de instrucciones como en el de datos resultarán en accesos para el segundo nivel, por otro lado cada reemplazo existente en

el cache DL1 exigirá una escritura en el cache de segundo nivel, pues como se utiliza la política *writeback*, el valor se encuentra actualizado únicamente en el cache de menor nivel y debe escribirse en las posiciones bajas de memoria. Dado que

$$\#Misses_{IL1} + \#Misses_{DL1} + \#Reemplazos_{DL1} = 2 + 63 + 47 = 112$$

, se darán 112 accesos al cache de segundo nivel (DL2). De esos accesos, los misses en primer nivel corresponden a lecturas (pues se están trayendo bloques de niveles menores de jerarquía) y los correspondientes a reemplazos resultan en escrituras en los niveles inferiores, por esta razón existen 47 escrituras y 65 lecturas sobre el cache L2.

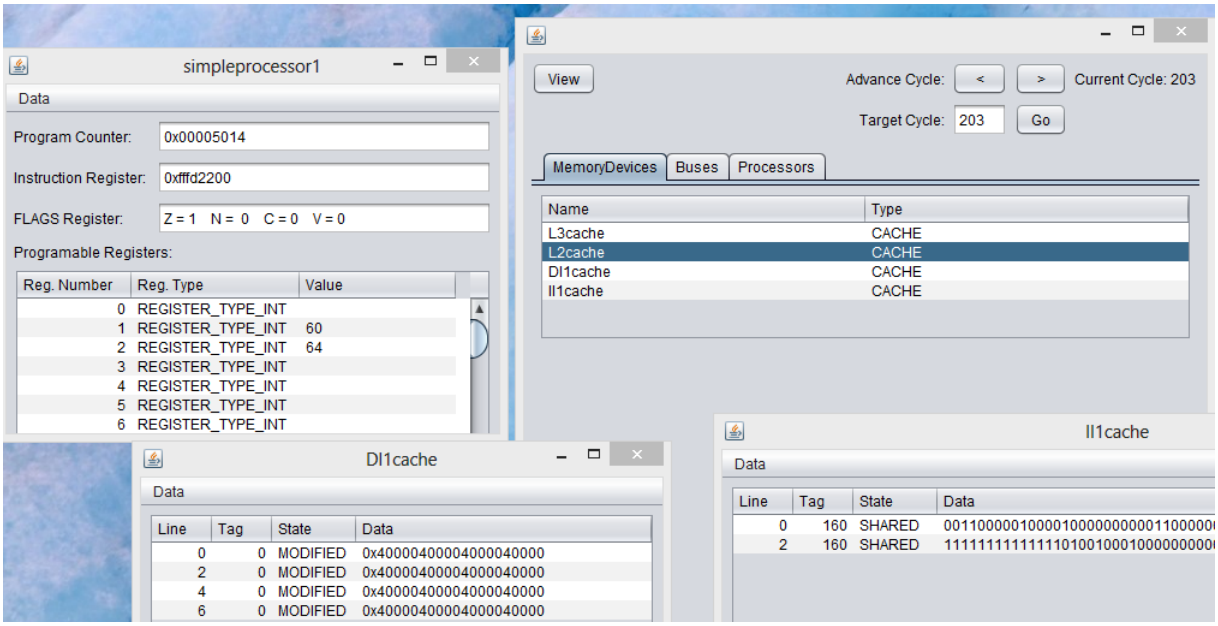


Figura 4.7: Captura de pantalla de visor de trazas Simco Viewer para el ejemplo de jerarquía de memoria

En la figura 4.7 se puede ver una captura de pantalla del visor de trazas, donde se evidencia cómo se podrían visualizar los diferentes componentes del sistema. Gracias a esta disposición, se muestra que SimCo podría ser utilizado a nivel educativo para apoyar la comprensión de los sistemas con cache (sea cual sea su configuración), permitiendo validar suposiciones al respecto de la variación del sistema en cada ciclo.

- Prueba de coherencia de cache para procesadores de memoria simétrica

  En esta prueba se mostrará la utilidad de SimCo para estudiar los algoritmos de coherencia de cache utilizados en procesadores de memoria simétrica. Además, se mostrará cómo configurar el simulador para pruebas de múltiples procesadores. Se simulará un multicore con dos núcleos de ejecución. La jerarquía de memoria utilizada

será la mostrada en la figura 4.8, a excepción del cache L3, que fue removido pues no aporta al ejemplo. Como núcleos de ejecución se utilizó la el CPU implementado 'SimpleUnipipedProcessor' y todas las interconexiones se realizaron con buses.

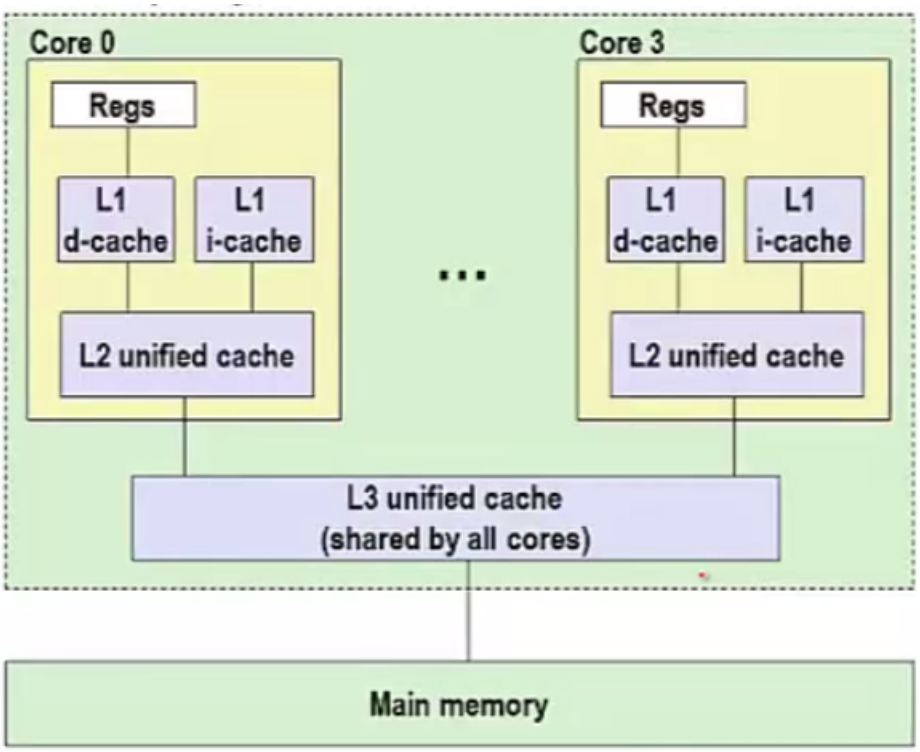


Figura 4.8: Jerarquía de memoria, procesador core I7, tomada de [43]

Cache L1 de instrucciones	
Sets:	8
Asociatividad:	2
Tamaño de línea:	16 bytes
Política de remplazo:	LRU

Cache L1 de datos	
Sets:	8
Asociatividad:	2
Tamaño de línea:	16 bytes
Política de remplazo:	LRU
Política de escritura:	writethrough

Cache L2 Unificada	
Sets:	16
Asociatividad:	4
Tamaño de línea:	32 bytes
Política de remplazo:	LRU
Política de escritura:	writeback
Protocolo de coherencia utilizado:	MSI

Como prueba se utilizó el siguiente programa:

```
 # Código del procesador 1
ORG 0x5000
# Iniciación de variables
ANDI R2, R2, 0 # se realiza la operación and de R2 con 0, cargando el valor
0 en R2
ANDI R1, R1, 0
LLO R2, 35
# Carga de dirección 0
SW R2, 0, R1
# Instrucciones de relleno para demorar
ADDI R2, R2, 1
ADDI R2, R2, 1
SW R2, 0, R1
HALT

# Código del procesador 2
ORG 0x6000
# Iniciación de variables
ANDI R2, R2, 0
ANDI R1, R1, 0
LLO R2, 10
ANDI R1, R1, 0
ANDI R1, R1, 0
LW R2, 0, R1
HALT
```

El primer procesador se configura para que inicie su ejecución en la dirección 0x5000, mientras que el segundo se configura para que lo haga en la 0x6000, de este modo, las instrucciones debajo de 'ORG 0x5000' [10] serán ejecutadas entonces por el primer procesador, mientras que las debajo de 'ORG 0x6000' serán ejecutadas por el segundo. El código de ambos procesadores es muy simple, consiste en realizar accesos de memoria a una dirección compartida de memoria. Se agregan algunas instrucciones de relleno para asegurar el orden esperado de los accesos. A continuación se indican los accesos a memoria realizados por el programa, así como los efectos a observarse en los caches y en el bus de memoria:

[10] La directiva ORG es típica de lenguajes ensambladores para indicar que las subsiguientes instrucciones se deberán colocar a partir de la dirección parámetro.

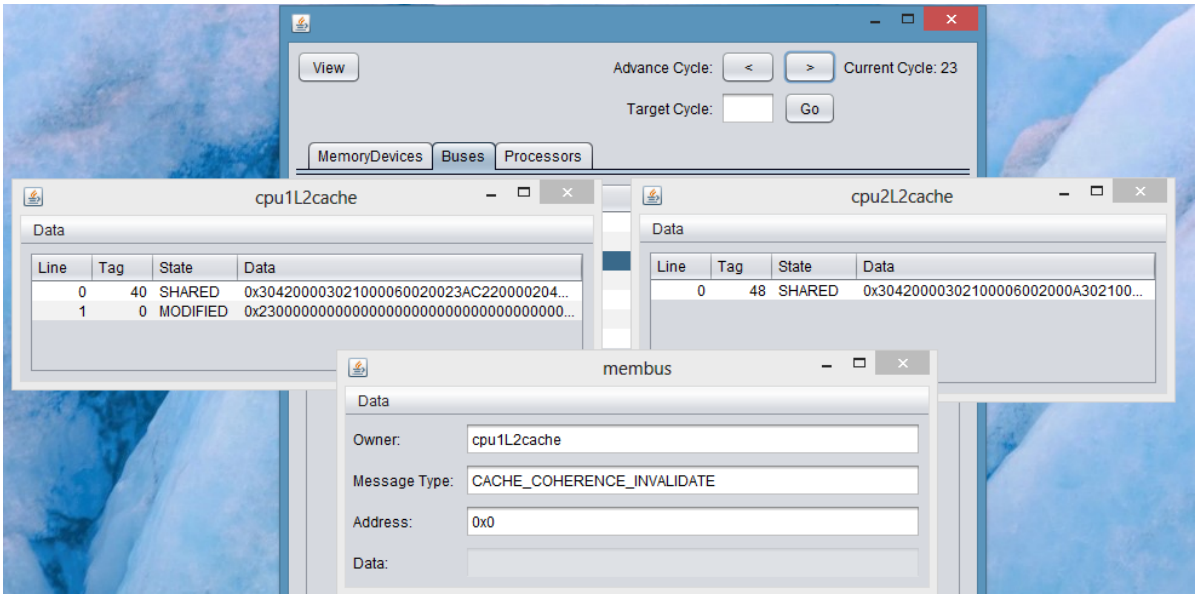


Figura 4.9: Escritura a dirección 0 por el procesador 1, por tanto, su estado pasa a ser modificado.

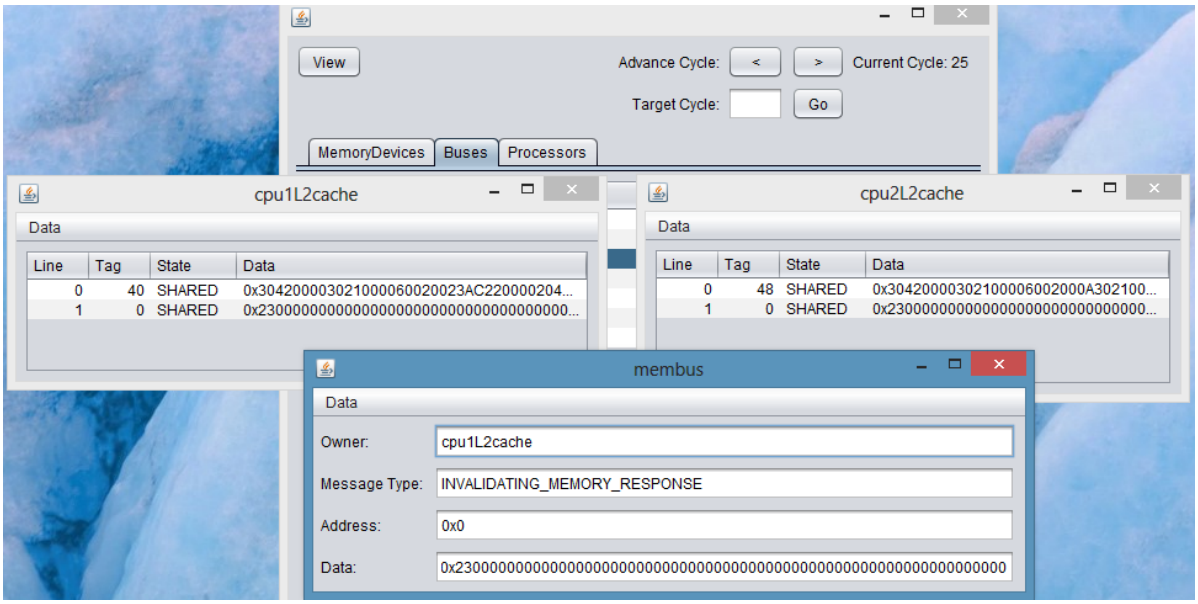


Figura 4.10: Lectura a dirección 0 por el procesador 2. Se provee el valor desde el cache del procesador 1, indicando que se debe invalidar el acceso al nivel inferior de la jerarquía. Ambas líneas pasan a estar en estado compartido.

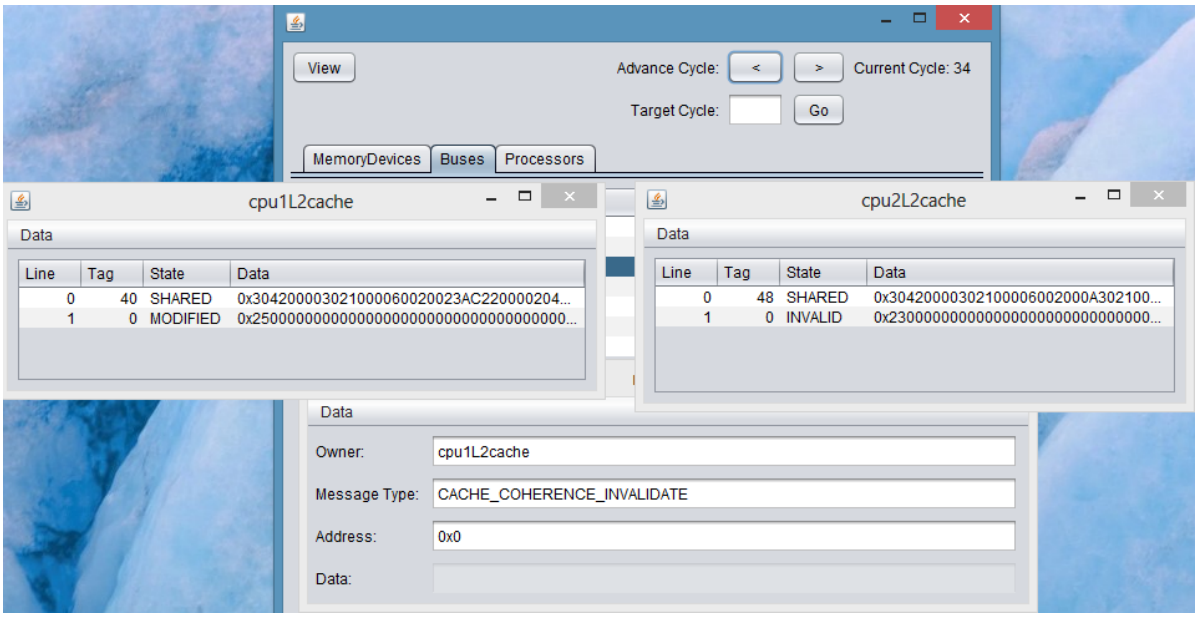


Figura 4.11: Escritura a dirección 0 por el procesador 1, su estado pasa a ser modificado y se envía un mensaje de invalidación hacia el resto de los caches.

Tiempo	Procesador 1	Procesador 2	Efecto esperado
t1	Store del valor 35 decimal a la dirección 0		Se guarda el valor 35 decimal en el cache L2 del procesador 1, dejando en estado modificado la línea correspondiente a ese valor.
t2		Load del valor contenido en la dirección 0	Se está intentando cargar el valor de una dirección, cuya copia más actualizada se encuentra en el cache L2 del procesador 1, por esta razón, el dato se proveerá desde ahí, cancelando el pedido hacia memoria principal y dejando ambas líneas en estado compartido (SHARED).
t3	Store del valor 40 decimal a la dirección 0		Como el bloque se encuentra compartido por varios dispositivos de memoria, al intentar realizar una escritura sobre este se deberá enviar un mensaje de invalidación al resto de los dispostivos.

Los resultados de esta prueba se pueden observar claramente en la aplicación SimcoViewer. En las figura 4.9, 4.10 y 4.11 se muestran las diferentes capturas de pantalla de la aplicación, donde se observan los resultados esperados indicados en la tabla anterior.

Con los resultados de esta prueba se pretende mostrar cómo SimCo podría ser utilizado para enseñar los algoritmos de coherencia de cache utilizados en multiprocesadores de memoria

compartida, así como la sobrecarga que demandan sobre la interconexión.

5. Conclusiones

Durante la realización de este proyecto se obtuvieron dos resultados fundamentales, los cuales cumplen con los objetivos planteados al inicio: en primer lugar, se realizó un estudio del estado del arte en simulación en arquitectura de computadoras, del cual se obtuvo como resultado una buena noción de qué se está estudiando en el área a nivel internacional. Se observó que los esfuerzos mayores están avocados hacia la elaboración de simuladores más rápidos, para facilitar la ejecución reiterada de benchmarks costosos durante la exploración de mejoras de diseño. Dichos simuladores incluyen interesantes ideas, como simuladores paralelizables [13], simulación estadística [18] [17], o simulación acelerada por FPGA [19]. Otras áreas activas de investigación son la simulación de unidades de procesamiento gráfico (GPU) y de consumo energético de los diferentes componentes (memorias, procesadores, etc).

Como resultado colateral del estudio, se logró una noción de cuáles son las propuestas actuales a nivel de diseño de arquitectura y microarquitectura en sistemas computacionales, lo cual permitirá en un futuro orientar de forma más precisa a aquellos estudiantes que deseen realizar investigación en el área de arquitectura de computadoras. Esto es un aporte al grupo de investigación MINA (Network Management - Artificial Intelligence) del Instituto de Computación, pues actualmente no se está investigando en el área de arquitectura de computadoras.

El proyecto produjo como resultado un producto de software: el simulador SimCo, el cual es un simulador de precisión de ciclo, orientado a ejecución, con diseño modular y con un enfoque educativo en su desarrollo y presentación. Al día de hoy, el simulador SimCo incluye 5608 líneas de código + 1351 de comentarios (sin incluir las pruebas). El conteo fue realizado con la herramienta CLOC [28]. Además se desarrolló el visor de trazas SimcoViewer, el cual consiste en una interfaz gráfica que permite explorar lo acontecido durante las simulaciones de forma ágil y simple. Los códigos de ambos programas se encuentran disponibles de forma pública en el repositorio del autor [24].

6. TRABAJO FUTURO

Las líneas de trabajo futuro serán presentadas en dos grandes grupos. En primer lugar están aquellas mejoras y optimizaciones que por ser de una carga de trabajo pequeña no ameritan un proyecto nuevo. En esta categoría hay varios puntos para trabajar, donde se destacan dos: mejorar el tiempo de ejecución de la aplicación, mediante la aplicación de técnicas de aceleración sugeridas a lo largo del código (por ejemplo, la utilización de pools [11] de objetos para los pedidos de memoria) e implementar la interfaz gráfica para las redes de interconexión basadas en switches.

En la segunda categoría de trabajos futuros se encuentran aquellas mejoras que por su gran volumen de trabajo ameritarían ser abordadas por otros proyectos de grado. Dentro de esta categoría, como primer continuación del presente trabajo se propone realizar la implementación de un CPU superescalar y realizar entonces la validación del simulador contra un hardware real. Otro proyecto de grado podría estar enmarcado en realizar incorporaciones de interés educativo del Instituto de Computación, en particular,„ el agregado de las arquitecturas utilizadas en las asignaturas del Depto. de Arquitectura, Sistemas Operativos y Redes de Computadoras, siendo x86 la arquitectura de mayor interés pues es utilizada en varios de estos cursos.

Para el interés del grupo de investigación MINA, las posibilidades con el simulador son bastante amplias. En el anexo se presenta un ejemplo de cómo podría ser utilizado el simulador en el contexto de un proyecto de investigación y se presenta una propuesta de diseño para explorar. Otra posible continuación de este proyecto a nivel de investigación se podría centrar en el estudio de las redes de interconexión de multiprocesadores, tanto de aquellas que se localizan dentro del mismo chip (NoC), como las que se encuentran fuera. Según lo investigado, al día de hoy se están realizando propuestas de mejores protocolos de enrutamiento, de detección de deadlock y livelock, de ahorro de energía, entre otras, y por tanto proveer a SimCo de una herramienta de modelado más poderosa para estas redes abriría puertas a más trabajos y posibles aportes.

Otra posible extensión podría estar enfocada en el estudio de procesadores heterogéneos y en la incorporación de estos a las herramientas de modelado provistas por SimCo, mediante la creación de una nuevas unidades de ejecución: una GPGPU y otra CPU que sea capaz de reenviar instrucciones hacia la GPGPU. Dicha incorporación debería estar acompañada de un estudio previo de este tipo de organizaciones.

Por último, dentro de las líneas de interés más lejanas a lo abordado por este proyecto se encuentra el estudio del uso energético del sistema computacional. Según lo estudiado,

[11] Un pool de objetos es un tipo abstracto de datos utilizado para evitar el uso intensivo de memoria dinámica en objetos con ciclo de vida corto. El pool mantiene una colección de un cierto tipo de objetos, los cuales están a disposición para cuando algún otro módulo los necesite. En ese momento, se solicita al pool un nuevo objeto y cuando se termina su utilización es devuelto a la estructura.

permitir a SimCo el modelado de dichas variables requiere del trabajo conjunto con el área de Ingeniería Eléctrica pero sería sin lugar a dudas de vital importancia si se desea realizar contribuciones de vanguardia en diseño de computadoras.

7. Apéndices

7.1. Protocolos de coherencia de cache

En el simulador SimCo se implementan a la fecha dos protocolos de coherencia de cache de husmeo:

1. Protocolo MSI: A cada línea del cache se le asocia un estado, el cual indica la relacion de los datos contenidos con respecto al resto de la jerarquía de memoria. Los estados posibles son:

   - Modificado (Modified - M): La línea contiene la copia más actualizada del bloque.
   - Compartido (Shared - S): La línea contiene un bloque que está compartido en líneas de otros caches y por lo tanto es una copia de sólo lectura.
   - Inválido (Invalid - I): El bloque contenido en esta línea está desactualizado y por lo tanto es inválido.

   Cuando se realiza un acceso a una cierta línea, según el estado en que esta se encuentre se deberán enviar mensajes por el bus para mantener la *coherencia* del estado de un cieto bloque. Por ejemplo, si un bloque se encuentra compartido (estado S) y este se escribe en uno de los caches, primero se deberá cambiar el estado de las demás copias a inválido.

2. Protocolo MESI: Similar al protocolo MSI pero con un estado extra (E - Exclusivo), el cual indica que el bloque no está modificado, pero sí que es la única copia del mismo que existe en los caches. La incorporación de este estado reduce el uso del bus común. [1]

7.2. Propuesta de investigación con SimCo

La definición de AMAT (average memory access time - AMAT) para una memoria cache es la siguiente:

$$AMAT = Hit\_time + Miss\_rate * Miss\_penalty$$

, donde *hit time* es el tiempo de acceso a memoria en caso de que el dato buscado se encuentre en el cache, *miss rate* es la fracción de accesos fallidos al cache y *miss penalty* es el tiempo extra que se debe invertir para obtener el dato en caso de que este no se encuentre en el cache.

De la definición de AMAT, reducir la penalización por fallos es una de las posibles técnicas para reducir el tiempo de acceso promedio a memoria. Las técnicas más simples se basan en un buen diseño del cache, como aumentar la asociatividad y el tamaño de los bloques, sin embargo, dichas técnicas tienen un cierto grado de aplicabilidad antes de volverse contraproducentes (por ejemplo, el aumento de la asociatividad aumenta el hit time [1]). Varias técnicas avanzadas han sido propuestas, siendo la incorporación de una cache de víctimas (*victim cache*), una de las más exitosas. Una cache de víctimas consiste en una pequeña memoria cache ubicada cerca del cache, de unas pocas líneas (4 u 8 típicamente), totalmente asociativa, y que guarda las distintas líneas que van siendo expulsadas del cache a causa de los reemplazos. Si se produce un fallo en un acceso, antes de solicitar el bloque al siguiente nivel de jerarquía, se verifica si el bloque buscado se encuentra en la cache de víctimas, como se muestra en la figura 7.1. El resultado visto de la cache de víctimas es que aumenta temporalmente la asociatividad de cualquier bloque durante el tiempo en que éste está siendo utilizado de forma intensiva.

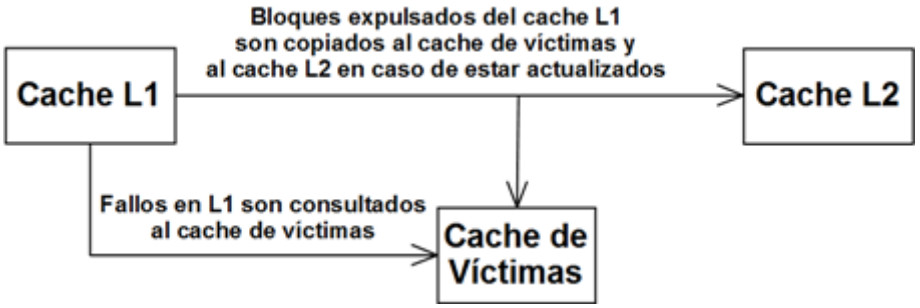


Figura 7.1: Organización de cache de víctimas

Siguiendo la línea de pensamiento de las cache de víctimas, se propone una mejora a aplicar para el caso de las líneas efímeras (*transient*). Una línea se denomina efímera mientras el bloque que la va a ocupar está siendo buscado desde memoria. Según la investigación realizada a comienzo del proyecto, no se encontró consenso al respecto de qué realizar en caso de obtener accesos a una línea con estas características (en [33] se consideran dichas posiciones de memoria como perdidas y se propone usarlas para otras funciones). Si se utiliza una política de remplazo LRU y se obtiene un acceso a una línea efímera, esto significa que el bloque que está por ser desplazado es ahora el más recientemente usado y por tanto no debería ser desplazado. Una mejora posible sería, en caso de obtener un acceso como el mencionado, realizar una nueva selección de línea a reemplazar e intercambiar el valor de la línea elegida con el de la línea efímera accedida. El simulador SimCo podría ser modificado levemente para soportar este cambio y verificar de este modo la aceleración obtenida en el AMAT con dicho agregado, ante un conjunto de benchmarks a seleccionar.

7.3. EJEMPLO DE ARCHIVO DE CONFIGURACIÓN DE SIMCO

En este archivo de configuración se configura un sistema con un cpu, un cache L1 de instrucciones/datos y una memoria RAM.

```
 cyclelimit = 100
isa = mips32
bus
     name = cpu1bus
     width = 4
end
bus
     name = membus
     width = 16
end
simpleunpipedprocessor
     name = simpleprocessor1
     meminterface = cpu1bus
     pcvalue = 5540
end
ram
     name = ram
     capacity = 8192
     ports = 1
     latency = 9
     interface = membus
end
cache
     name = l1cache
     setcount = 8
     associativity = 2
     linesize = 16
     replpolicy = lru
     writepolicy = writeback
     coherence = msi
     ports = 1
     latency = 1
     upperinterface = cpu1bus
     lowerinterface = membus
end
```

REFERENCIAS

[1] Computer Architecture, A Quantitative Approach. John L Hennessy, David A.Patterson. http://booksite.mkp.com/9780123838728/

[2] IEEE International Sympossium on High Performance Computer Architecture (HPCA). Repository http:// www.hpcaconf.org/

[3] The Annual IEEE/ACM International Symposium on Microarchitecture. http://www.microsymposia.org/

[4] Definición de computadora: es.wikipedia.org/wiki/Computadora. Último acceso 10/06/2014

[5] W. J. Dally and B. Towles. Principles and practices of interconnection networks. Morgan Kaufmann, 2003.

[6] International Symposium on Computer Architecture. Ediciones 2014 y anteriores: http://cag.engr.uconn.edu/isca2014/previous.html

[7] International Conference on Computer Design. Ediciones 2014 y anteriores: http://www.iccd-conf.com/ICCD_Previous_Editions.html

[8] IEEE International Symposium on Performance Analysis of Systems and Software. Ediciones 2014 y anteriores: http://www.ispass.org/ispass2014/

[9] J. J. Yi, L. Eeckhout,D. J. Lilja, B. Calder, L. K. John, and J. E. Smith. “The Future of Simulation: A Field of Dreams,” IEEE Computer, 39(11):22–29, Nov. 2006.

[10] T. Austin, E. Larson, and D. Ernst, “SimpleScalar: An Infrastructure for Computer System Modeling,” IEEE Computer, vol. 35, no. 2, pp. 59–67, 2002. http://www.simplescalar.com/

[11] K. Skadron et al., “Challenges in Computer Architecture Evaluation,” Computer, Aug. 2003, pp. 30-36.

[12] Charles Price. MIPS IV Instruction Set, revision 3.1. MIPS Technologies, Inc., Mountain View, CA, January 1995.

[13] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep and Anant Agarwal "Graphite: A Distributed Parallel Simulator for Multicores"The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA), Jan 2010

[14] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, Analyzing CUDA workloads using a detailed GPU simulator. IEEE International Symposium on Performance Analysis of Systems and Software, April 2009.

[15] V. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and E. E. ATTILA: a cycle-level execution-driven simulator for modern GPU architectures. Int'l Symp. on Performance Analysis of Systems and Software, pages 231–241, March 2006.

[16] Yi Yang, Ping Xiang, Mixe Mantor, Huiyang Zhou: CPU-assisted GPGPU on fused CPU-GPU architectures. In: Proceedings of the 2012 IEEE 18th International Symposium on High Performance Computer Architecture HPCA '12 (2012)

[17] Ehsan K. Ardestani and Jose Renau, 'ESESC: A Fast Multicore Simulator Using Time Based Sampling' The 19th IEEE International Symposium on High Performance Computer Architecture (HPCA), 2013

[18] S. Nussbaum and J. Smith, "Modeling Superscalar Processors via Statistical Simulation," Proc. 11th Ann. Int'l Conf. Parallel Architectures and Compilation Techniques, IEEE CS Press, 2001, pp. 15-24.

[19] Chiou, Derek, et al. "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators."Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture. IEEE Computer Society, 2007.

[20] Raghavan, Arun, et al. Çomputational sprinting."High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on. IEEE, 2012.

[21] EPIC: Explicitly parallel instruction computing. Entrada de Wikipedia: http://en.wikipedia.org/wiki/Explicitly_parallel_instruction_computing

[22] Simulation Modelling with Pascal, Davies R. and O'Keefe R., Prentice Hall, ISBN 013811571-0, 1989.

[23] Repositorio de SimCo: https://github.com/federivero/SimCo

[24] Repositorio GitHub de Federico Rivero: https://github.com/federivero

[25] Haskell. The Craft of Functional Programming. Third Edition. Simon Thompson. Addison-Wesley, ISBN 0-201-34275-8. Online: http://www.haskellcraft.com/craft3e/Home.html

[26] Formato Properties. Entrada en wikipedia: http://en.wikipedia.org/wiki/.properties

[27] Desikan, Rajagopalan, Doug Burger, and Stephen W. Keckler. "Measuring experimental

error in microprocessor simulation."Proceedings of the 28th annual international symposium on Computer architecture. ACM, 2001.

[28] Herramienta CLOC - Count Lines of Code. Online: http://cloc.sourceforge.net/

[29] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator. http://sesc.sourceforge.net. 2005.

[30] Arquitectura MIPS32. Especificación de la universidad de Waterloo. Disponible online: https://www.student.cs.uwaterloo.ca/ isg/res/mips/opcodes

[31] Aspectos Avanzados de Arquitecturas de Computadoras. Moodle: https://eva.fing.edu.uy/course/view.php?id=268

[32] Sistema AS/400. Entrada en Wikipedia http://es.wikipedia.org/wiki/AS/400

[33] González, Antonio, Fernando Latorre, and Grigorios Magklis. "Processor microarchitecture: An implementation perspective."Synthesis Lectures on Computer Architecture 5.1 (2010): 1-116.

[34] Loh, Gabriel H., Samantika Subramaniam, and Yuejian Xie. "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration."Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on. IEEE, 2009.

[35] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A full system simulator for x86 CPUs. In Proceedings of the 2011 Design Automation Conference, June 2011.

[36] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Werner, B. (2002). Simics: A full system simulation platform. Computer, 35(2), 50-58.

[37] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve, “Rsim: Sim- ulating shared-memory multiprocessors with ilp processors,” Computer, vol. 35, no. 2, pp. 40–49, 2002.

[38] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. "Simflex: Statistical sampling of computer system simulation". IEEE Micro, 26:18–31, July 2006.

[39] Binkert, Nathan, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness et al. ’The gem5 simulator.’ ACM SIGARCH Computer Architecture News 39, no. 2 (2011): 1-7.

[40] Binkert, N. L., Dreslinski, R. G., Hsu, L. R., Lim, K. T., Saidi, A. G., and Reinhardt, S. K.

The M5 Simulator: Modeling Networked Systems. IEEE Micro 26, 4 (Jul/Aug 2006), 52-60.

[41] Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D., and Wood, D. A. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. SIGARCH Comput. Archit. News 33, 4 (2005), 92-99.

[42] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual", disponible en Web: http://www.intel.com/design/processor/manuals/253668.pdf.

[43] "The Hardware/Software Interface". Curso online de la University of Washington sobre arquitectura de computadoras. Sitio web: https://www.coursera.org/course/hwswinterface