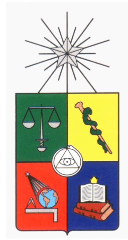


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

# ANÁLISIS E IMPLEMENTACIÓN DE SISTEMA DE MEMORIA DISTRIBUIDO PARA EL PROCESO DE CIRCUITOS INTEGRADOS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRICISTA

NICOLÁS ULRIKSEN PALMA

PROFESOR GUÍA:
VÍCTOR GRIMBLATT HINZPETER

MIEMBROS DE LA COMISIÓN:
HÉCTOR AGUSTO ALEGRÍA
GERARDO LEÓN MARTÍNEZ

SANTIAGO DE CHILE
2014

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRICISTA
POR: NICOLÁS ULRIKSEN PALMA
FECHA: 2014
PROF. GUÍA: VÍCTOR GRIMBLATT HINZPETER

ANÁLISIS E IMPLEMENTACIÓN DE SISTEMA DE MEMORIA DISTRIBUIDO PARA EL PROCESO DE CIRCUITOS INTEGRADOS

En la actualidad las personas ocupan una gran cantidad de dispositivos electrónicos, como computadores, teléfonos, relojes, refrigeradores e incluso automóviles, que contienen "chips" o circuitos integrados cada vez más complejos y con una mayor cantidad de componentes, lo cual involucra que las herramientas de diseño, implementación y construcción tengan que ir evolucionando de manera tal, que puedan estar al día con la creciente demanda de los circuitos integrados.

En la presente Memoria, realizada en la empresa Synopsys®, se analizan distintos tipos de sistemas de memoria distribuida, y su implementación en el software PUMA®, desarrollado por la misma empresa. Este software procesa principalmente los archivos de *máscaras* de los circuitos integrados, para convertirlos en un formato legible para las máquinas que los fabrican. El procesamiento de estos archivos se realiza simultáneamente en muchas máquinas (llamado *cluster*), que ocupan discos duros como medio de almacenamiento. Dado que este tipo de almacenamiento produce en el proceso "cuellos de botella", es el objetivo de este trabajo modificarlo por un sistema de memoria RAM compartida por todas las máquinas, para aumentar así la eficiencia del software PUMA®.

El sistema de almacenamiento se implementa con una librería propia de Synopsys®, que tiene la capacidad de leer y escribir datos entre máquinas. Mejorando el rendimiento, en el proceso de escritura y lectura, de archivos que van de los cientos de megabytes, hasta los cientos de gigabytes en el caso de procesos comerciales.

De las pruebas realizadas, se logra en el mejor de los casos, una reducción en el tiempo de ejecución de hasta un 10 %, realizadas bajo condiciones normales de procesamiento del *cluster*, y sin ningún agente externo perturbando la muestra. Finalmente, hay que indicar que se pudo mostrar que, si los archivos de entrada no poseen un determinado tamaño mínimo, los tiempos de ejecución aumentan y no se aprovecha la modificación del sistema de almacenamiento.

A Pablo y Carmen Gloria.

# Agradecimientos

Agradezco a Synopsys® por el apoyo dado para realizar la memoria, en especial a los profesionales que participaron directamente en su desarrollo.

Agradezco a mi madre, por el apoyo incondicional y constante desde el primer año en la universidad. A mi padre, por impulsarme en el camino que seguí.

Agradezco a todos los profesores que a lo largo de la carrera, me han formado como profesional.

Agradezco a todos mis amigos y compañeros, que han estado en el camino apoyando y motivando en los momentos buenos y malos.

# Tabla de Contenido


**1. Introducción 5**

1.1. Motivación . . . 6

1.2. Alcances y Objetivo General . . . 7

1.3. Objetivos Específicos . . . 7

1.4. Estructura de la Memoria . . . 8

**2. Antecedentes 9**

2.1. Circuito Integrado . . . 9

2.2. Diseño de Circuitos Integrados . . . 10

2.3. Preparación de Datos de Máscara . . . 13

2.4. Fotolitografía . . . 13

2.5. Sistemas de Memoria Distribuida . . . 14

2.5.1. Sistema Memcached . . . 14

2.6. Base de Datos en Memoria . . . 15

2.6.1. Sistema de Almacenamiento Redis . . . 15

2.7. Tcl: Lenguaje de Herramientas de Comando . . . 16

**3. Análisis e Implementación 17**

3.1. Análisis . . . 17

3.2. Implementación . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.2.1. Secuencia de Datos o Streams . . . . . . . . . . . . . . . . . . . . . . . 20
3.2.2. Atomicidad de Operaciones . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.3. Implementación para Múltiples Máquinas . . . . . . . . . . . . . . . . . . . . . 21
**4. Discusión de Resultados** **22**
4.1. Comparación de Datos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.2. Resultados Preliminares . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.3. Resultados . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
**5. Conclusiones** **26**
**Bibliografía** **28**
**A. Apéndice A** **I**

# Índice de figuras

2.1. Pasos principales en el flujo de diseño IC . . . . . . . . . . . . . . . . . . . . . . 12

2.2. Esquema de un circuito integrado . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.3. Esquema de un sistema virtual de memoria compartida . . . . . . . . . . . . . . 14

4.1. Resultados preliminares con CDSL . . . . . . . . . . . . . . . . . . . . . . . . . . 23

4.2. Resultados preliminares de dp con CDSL archivo de entrada 70 kilobytes . . . . 24

4.3. Resultados en dp con CDSL archivo de entrada 468 megabytes . . . . . . . . . . 25

# Índice de tablas

A.1. Datos de archivo de entrada grande sin CDSL en múltiples máquinas . . . . . . I

A.2. Datos de archivo de entrada grande con CDSL en múltiples máquinas . . . . . I

A.3. Datos de archivo de entrada pequeño sin CDSL en múltiples máquinas . . . . . II

A.4. Datos de archivo de entrada pequeño con CDSL en múltiples máquinas . . . . II

A.5. Datos de archivo de entrada pequeño sin CDSL en una sola máquina . . . . . . II

A.6. Datos de archivo de entrada pequeño con CDSL en una sola máquina . . . . . III

# Capítulo 1

# Introducción

En la actualidad las personas ocupan una gran cantidad de dispositivos electrónicos, ya sea computadores, teléfonos, relojes, refrigeradores e incluso automóviles, que contienen “chips” o circuitos integrados, cuyos componentes internos son de tamaño nanométrico, generalmente fabricados con fotolitografía, y que a través del tiempo han ido avanzado en complejidad y tamaño. El proceso de diseño de estos circuitos ha pasado de ser algo que se diseñaba a mano, a necesitar software especializado que sea capaz de procesar todos los componentes que poseen.

Así mismo, el software que se ocupa para diseñar y crear los chips, también tiene que ir avanzando en su capacidad de manejar cada vez una mayor cantidad de transistores por circuito integrado. Esto genera que el volumen de datos que se tiene que procesar no pueda ser manejado por un solo computador, sino por un grupo de ellos, llamado *cluster*, que trabajan en conjunto para poder procesar el gran volumen de datos. También se necesita ir actualizando las tecnologías ocupadas en el software, para ocupar en su total capacidad, los computadores cada vez más potentes disponibles en el mercado.

Esta memoria se realiza con el apoyo de la empresa Synopsys®, dentro de un contexto de análisis e implementación de una nueva tecnología que mejore el rendimiento del software que ayuda en la fabricación de circuitos integrados. Específicamente la modificación en el tipo de almacenamiento ocupado por el software llamado PUMA®, desarrollado por Synopsys®, pasando de un almacenamiento en base a disco duro, a uno en base a la memoria RAM de los computadores del *cluster*, con el objetivo de aumentar el rendimiento actual de PUMA® en el procesamiento de archivos con datos de circuitos integrados. Teniendo en cuenta que en este trabajo de Memoria se utiliza propiedad intelectual perteneciente a Synopsys®, no se muestran los códigos pertenecientes a la implementación misma.

## 1.1. Motivación

Hoy en día, la producción de circuitos integrados en el mundo es tan diversa como importante para la calidad de vida actual, desde procesadores para los computadores, hasta pequeños chips en autos de juguetes. Esta gran demanda de diseño y fabricación de circuitos integrados, no podría ser satisfecha si no existiera automatización y apoyo de software desde el ámbito de la producción de los circuitos integrados.

Una de las empresas en el mercado actual, que desarrollada productos de automatización del diseño es Synopsys®, una empresa internacional que posee una gama inmensa de productos para cada punto de la fabricación de un circuito integrado. Un flujo de diseño simple, para la generación de un circuito integrado sería:

1. Especificar el chip
2. Generar las compuertas lógicas
3. Hacer el chip verificable
4. Verificar que las compuertas lógicas funcionen correctamente
5. Generar el esquema del chip
6. Verificar el esquema
7. Pasar a silicio

El *esquema* de un chip, es una representación de cómo y dónde se dispondrán los diferentes elementos que lo conforman. Es un plano de cómo tiene que ser construido posteriormente el circuito.

Para realizar el último punto, son necesarios una serie de pasos importantes para la fabricación del circuito integrado. En estos pasos se generan los datos necesarios para su fabricación, transformando las estructuras complejas del circuito integrado en figuras simples, las cuales son leídas por la maquinaria que hace la fabricación en masa. Este paso es costoso en tiempo y espacio, llegando a ocupar cientos de gigabytes en disco duro y cientos de procesadores.

El programa que realiza estas tareas actualmente es CATS®, el cual fue desarrollado hace más de 20 años, cuando las características de los circuitos integrados no eran tan

masivas como las actuales en cantidad de transistores y en tamaño de las conexiones de cobre.

Durante su operación, CATS® escribe estados intermedios a disco duro. Ésto genera que múltiples procesos, en distintos nodos del *cluster*, se ejecuten de manera más lenta que si se aprovecharan las características de los *clusters* actuales con grandes cantidades de memoria RAM disponible, que tiene lectura y escritura más rápida que el disco duro, y una infraestructura de red gigabit.

## 1.2. Alcances y Objetivo General

El objetivo general del trabajo de Memoria, es crear un prototipo funcional, donde se puedan realizar pruebas de rendimiento en *clusters* reales, y así comparar diferentes arquitecturas de soluciones, y también con la solución actual, esperando lograr un rendimiento mejor.

La solución propuesta, es ocupar un sistema de memoria compartida distribuida, que es equivalente a generar una memoria RAM virtual, accesible desde cualquier punto de un *cluster*. Esta propuesta se implementa en forma de prototipo sobre el software PUMA®.

## 1.3. Objetivos Específicos

Los objetivos específicos son los siguientes:

1. Investigar y encontrar diferentes tecnologías que permitan solucionar el problema descrito anteriormente y cumplir con los requisitos dados por la empresa Synopsys®.
2. Verificar la factibilidad de cada solución encontrada, según los requerimientos dados por la empresa, y descartar las no adecuadas.
3. Determinar y dividir las diferentes funcionalidades a modificar en el sistema original, y encontrar las partes del código fuente que deberán ser modificadas.
4. Implementar la solución provisoria en una de las funcionalidades de la arquitectura PUMA, para probar el uso del sistema de memoria distribuido.
5. Definir la interfaz de comunicación entre arquitectura PUMA y sistema de memoria.

6. Implementar la solución que responda a la interfaz definida en el punto anterior, y que abarque todas las funcionalidades detectadas en el objetivo 3.

## 1.4. Estructura de la Memoria

El resto del trabajo de Memoria está estructurado de la siguiente manera:

- **Capítulo 2. Antecedentes:** Corresponde a la revisión bibliográfica o antecedentes. En este capítulo se explican los conceptos necesarios para la comprensión y contextualización del trabajo.
- **Capítulo 3. Análisis e Implementación:** Corresponde a la implementación de software realizada. En este capítulo se describen los pasos seguidos para llevar a cabo la implementación.
- **Capítulo 4. Discusión de Resultados:** Corresponde a los resultados obtenidos. En este capítulo se presentan y discuten los resultados obtenidos en las pruebas realizadas.
- **Capítulo 5. Conclusiones:** Se enumeran las conclusiones del trabajo realizado y se proponen tareas para el futuro.

# Capítulo 2

# Antecedentes

El presente capítulo tiene por objetivo ubicar al lector en el entorno en el cual se desarrolla este trabajo de Memoria, entregando los antecedentes previos y necesarios para su contextualización.

## 2.1. Circuito Integrado

Un circuito integrado (IC, por su sigla en inglés), o circuito integrado monolítico, es un conjunto de circuitos eléctricos en una base pequeña, normalmente de silicio, llamada "chip". Los circuitos integrados son usados en casi todo los equipos de hoy en día y han revolucionado el mundo de la electrónica. Los IC pueden ser muy compactos, teniendo hasta muchos miles de millones de transistores y otros componentes eléctricos en un área menor a $1[cm^2]$. El ancho de cada línea de conducción en un circuito, se ha ido reduciendo a medida que avanza la tecnología, bajando de $100[nm]$ el año 2008, hasta llegar a las decenas en el año 2013.

La integración de grandes cantidades de pequeños transistores en un solo "chip", fue un gran avance en el ensamblaje de circuitos electrónicos discretos, en los cuales cada componente está separado del resto, siendo soldadas en una base común. Hay dos ventajas principales de los IC sobre los circuitos discretos: costo y rendimiento. El costo es menor, dado que todos sus componentes son impresos como una sola unidad, ocupando procesos de litografía en vez de construir un transistor por vez. Respecto al rendimiento, éste es alto debido a que los transistores pueden cambiar de estado rápidamente y consumen mucho menos potencia, dado al reducido tamaño de los componentes y la proximidad entre ellos.

## 2.2. Diseño de Circuitos Integrados

El diseño de circuitos integrados, es una sub-área de la ingeniería eléctrica, que incluye la lógica y técnicas de diseño de circuito requeridas para diseñar IC. El diseño de IC puede ser separado en dos grandes áreas: digital y análogo. El diseño digital se basa en niveles discretos de voltaje; se ocupa para producir microprocesadores, FPGAs (Arreglo de Campo de Compuertas Programables), memorias RAM, ROM y ASICs (Circuito Integrado para Aplicaciones Específicas) digitales. Se enfoca en la correctitud de la lógica, maximizar densidad del circuito, y ubicar circuitos de manera que las señales sean encaminadas eficientemente.

Por otro lado el diseño analógico, se enfoca en la variación continua de voltaje, por tanto, son más complejos de diseñar y son más susceptibles al ruido, lo que provoca que tengan menor precisión que los digitales. Este diseño se ocupa para generar circuitos como op-amps, reguladores lineales, osciladores y filtros activos. Se centra más en la parte física de los componentes, tal como ganancia, disipación de potencia y resistencia (ver Lavagno, Martin y Scheffer, 2009 [4]).

Como se especifica en el trabajo Baker, 2010 [2], los pasos principales de un flujo de diseño de IC son:

1. **Especificar el chip:** Describe lo que el chip debería hacer. Estas especificaciones se escriben ocupando un lenguaje especial, como SystemVerilog o VHDL. Las especificaciones del chip, logran que éste tenga un objetivo claro y restricciones en su diseño; su funcionalidad (qué es lo que hará); el rendimiento del chip, tanto de velocidad como de consumo de energía; las restricciones tecnológicas, como el tamaño y el espacio disponible; y la tecnología de fabricación que se ocupará.

2. **Generar las compuertas lógicas:** En este paso se genera la lógica detallada de los *gates*, o compuertas del circuito. Se sintetiza el código del punto anterior (en SystemVerilog o VHDL), y se convierte a una red de compuertas lógicas, que es un esquema del circuito que responde a los requerimientos dados.

3. **Hacer el chip *testeable*:** Se generan datos y compuertas adicionales, para ayudar al fabricante a encontrar defectos en el chip. Además se agregan compuertas para mandar información hacia afuera del chip.

4. **Verificar que las compuertas lógicas funcionen:** Se verifica que las compuertas lógicas se comporten según como se diseñaron.

5. **Generar el *layout* o esquema del chip:** Se genera un *plano* del circuito, donde las compuertas lógicas se conectan de acuerdo al diseño y cumple con las restricciones impuestas. Muestra dónde estarán físicamente las compuertas lógicas y cómo serán conectadas por los cables.

6. **Verificar el esquema:** Se verifica nuevamente el circuito, el largo de las conexiones cambia la velocidad con que funciona el circuito. Se ejecutan verificaciones después del posicionamiento de las compuertas lógicas.

7. **Pasar a silicio:** Se produce la data del *plano* del circuito. Se hacen ajustes de imagen, según el tamaño de los semiconductores ocupados en las compuertas lógicas.

Una representación de estos pasos, se muestra en la figura 2.1.

Después del punto (5) se obtiene el esquema del chip, que contiene el plano de los componentes físicos y muestra cómo se agrupan y distribuyen en el espacio designado. Un ejemplo de este plano se puede observar en la figura 2.2.

En el punto (7), después de haber pasado las últimas verificaciones, se ejecutan una serie de pasos previos que son muy importantes. Se hace la corrección de imagen del esquema, lo cual implica pasar los datos a un formato manipulable por las máquinas fabricadoras de chip, puesto que estas máquinas tienen restricciones del tipo de archivo que pueden leer, vale decir, se necesita transformar los archivos de los layout al formato legible. Por esta razón se realizan diversas operaciones sobre el diseño final, entre ellas está fragmentar a un conjunto de pequeños rectángulos y trapezoides, escalar los datos, realizar operaciones *booleanas*, remover *overlaps* de las figuras, rotar, cambiar el tamaño de figuras, etc.

Los pasos descritos anteriormente son costosos, tanto en tiempo como en espacio, ya que se utilizan *clusters* de computadores para realizar los cálculos, lo cual puede durar, en el peor caso, días. En este proceso, el tamaño usual de los archivos de entrada es del orden de los 30 gigabytes, y al terminar su procesamiento pueden éstos llegar incluso a tamaños superiores a 150 gigabytes.

Figura 2.1: Pasos principales en el flujo de diseño IC

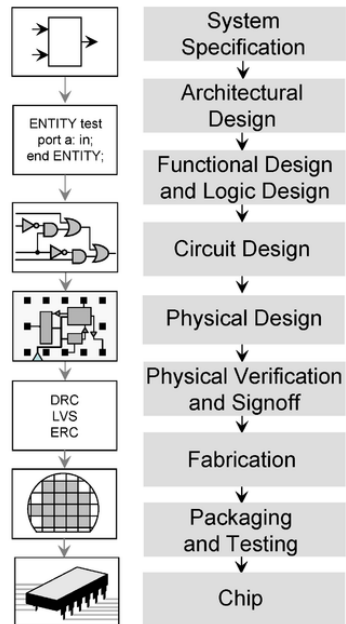
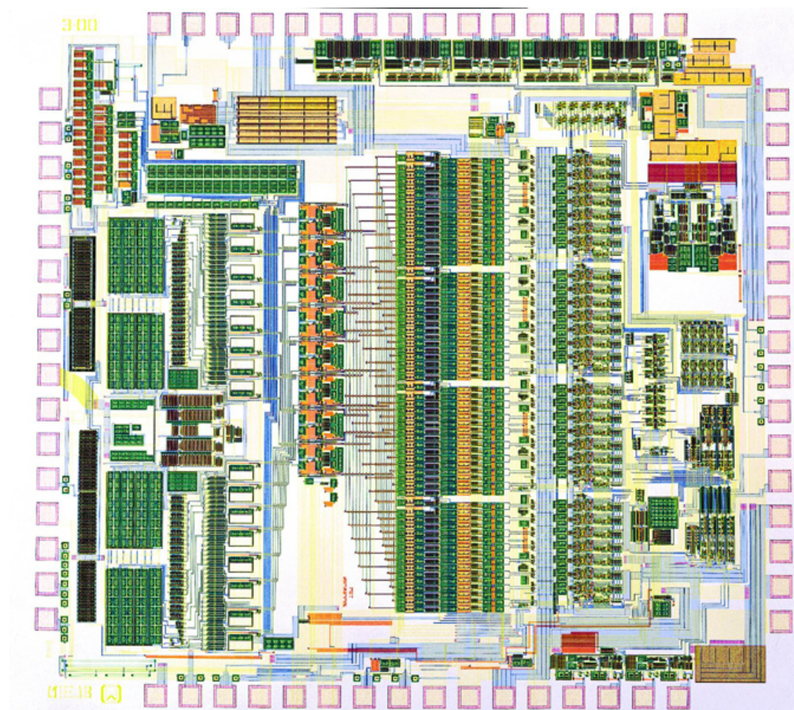



Figura 2.2: Esquema de un circuito integrado



## 2.3. Preparación de Datos de Máscara

En el último paso antes de pasar el IC a silicio, se genera la preparación de los datos de la máscara (MDP, por sus siglas en inglés), que es el proceso de traducir un archivo que contiene el conjunto de polígonos del esquema de un IC, a un conjunto de instrucciones que una máquina generadora de máscaras fotográficas pueda entender. La MDP usualmente involucra fractura de máscara, en el cual los polígonos complejos son modificados a figuras más simples, normalmente rectángulos o trapezoides.

## 2.4. Fotolitografía

La fotolitografía es un proceso empleado en la fabricación de dispositivos semiconductores o circuitos integrados. El proceso consiste en transferir un patrón desde una fotomáscara (denominada retícula), a la superficie de una oblea. El silicio, en forma cristalina, se procesa en la industria en forma de obleas. Las obleas se emplean como sustrato litográfico, no obstante existen otras opciones como el vidrio, el zafiro, e incluso metales. La fotolitografía (también denominada “microlitografía” o “nanolitografía”), trabaja de manera análoga a la litografía empleada tradicionalmente en los trabajos de impresión, y comparte algunos principios fundamentales con los procesos fotográficos.

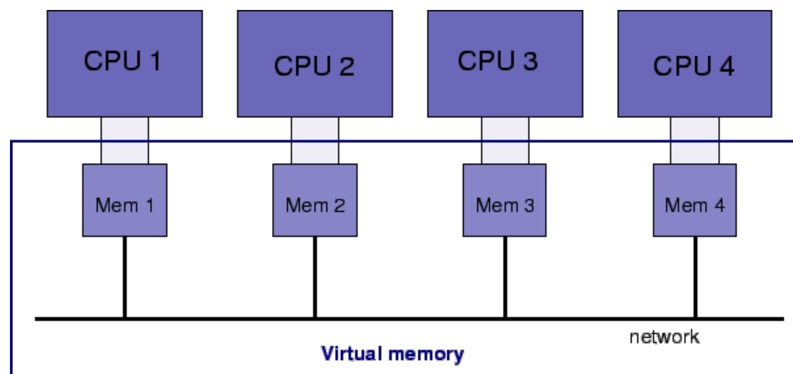
## 2.5. Sistemas de Memoria Distribuida

Sistemas de Memoria Distribuida o DSM, en Arquitectura de Computadores, es una forma de arquitectura de memoria donde éstas están físicamente separadas y pueden ser accedidas como una sola unidad lógica. Ésto permite a usuarios finales, acceder a datos compartidos sin usar procesos de intercomunicación. En otras palabras, el objetivo de un DSM es hacer transparentes las comunicaciones entre procesos (ver Patterson, David A. and John L. Hennessy, 2007 [5]).

Desde un punto de vista de programación, existen dos aproximaciones: memoria virtual compartida y objetos DSM. En el caso del primero, la idea principal es agrupar toda la memoria distribuida en un único direccionamiento, emulando una memoria RAM común y corriente. En el segundo caso, los datos compartidos son *objetos*, que son variables con funciones de acceso. De esta manera el usuario tiene que definir qué datos, es decir, qué objetos serán compartidos.

Un esquema de memoria virtual compartida se puede ver en la figura 2.3.

Figura 2.3: Esquema de un sistema virtual de memoria compartida




### 2.5.1. Sistema Memcached

En el área de la computación, se entiende como Memcached un sistema de *caching* de memoria distribuida con propósito general, siendo *caching* un componente que almacena información de recuperación rápida en sus usos futuros. Es generalmente usada para aumentar la velocidad de páginas web con contenido dinámico, que utilizan bases de datos, almacenando información y objetos en memoria RAM, para reducir el número de llamadas de lecturas a fuentes de datos externos (como lo son las bases de datos o una API, del inglés

*Application Programming Interface*).

La API de Memcached provee una tabla de *hash* (una estructura de datos que asocia llaves con valores) distribuida a lo largo de múltiples máquinas. Cuando ésta se llena, las inserciones siguientes causan que los datos más antiguos, sean eliminados en orden de antigüedad (sigla en inglés LRU, ver Brad Fitzpatrick, 2004 [3]). Las aplicaciones que ocupan Memcached, normalmente lo utilizan como una capa entre la aplicación y las formas de almacenamiento más lentas.

## 2.6. Base de Datos en Memoria

Una base de datos en memoria, o IMDB (del inglés, in-memory database), es un sistema de administración que se basa en la memoria principal (RAM) para almacenar los datos, y se contrasta con otros sistemas de administración de base de datos, que ocupan mecanismos de almacenaje en discos. Las IMDB son más rápidas que las bases de datos optimizadas para discos, dado que los algoritmos de optimización internos de las IMDB son más simples y ejecutan menos instrucciones de CPU. Otra ventaja, es que el acceso a datos en memoria principal, elimina los tiempos de búsqueda de los discos, lo que permite un mejor rendimiento y una mejor predicción de los tiempos de búsqueda.

### 2.6.1. Sistema de Almacenamiento Redis

Redis es un sistema de almacenamiento llave-valor en memoria *open source*, escrito en ANSI C. En su capa de abstracción más externa, el modelo de datos es un diccionario que *mapea* las llave con sus respectivos valores. Una de las principales características que lo diferencian de otros almacenamientos estructurados, es que no solo soporta *strings* sino que también otros tipos de datos abstractos como:

- Listas de *strings*
- Conjuntos de *strings*
- Conjuntos ordenados de *strings*
- Tablas de *hash* donde las llaves y los valores son *strings*

El tipo de valor determina qué operación está disponible para el mismo. Redis soporta operaciones atómicas, de alto nivel del lado del servidor, como intersecciones, uniones y

diferencias entre conjuntos, listas ordenadas y conjuntos ordenados (ver Jeremy Zawodny, 2009 [6], `http://redis.io` [1]).

## 2.7. Tcl: Lenguaje de Herramientas de Comando

El lenguaje de herramientas de comando o Tcl (de su sigla en inglés), es un lenguaje de *script* que posee una sintáxis sencilla para facilitar su aprendizaje, sin quitar funcionalidad y expresividad. Se usa principalmente en el desarrollo de prototipos rápidos, interfaces gráficas y en tests de software. Es usado en plataformas de sistema embebido, tanto en su totalidad o con funcionalidades disminuidas.

Es un lenguaje multiplataforma (Windows, Unix, Linux, Macintosh, etc.), que posee un sistema de objetos, basado en clases, totalmente dinámico. Todo en el lenguaje puede ser dinámicamente redefinido y modificado. Todos los tipos de datos pueden ser tratados como *strings*, incluido el código fuente, y posee una integración por medio de una interfaz al software gráfico Tk.

# Capítulo 3

# Análisis e Implementación

En el presente capítulo se analizan los requerimientos necesarios para la modificación del software, el diseño necesario de los algoritmos y su implementación.

## 3.1. Análisis

El software ocupado para la implementación de la memoria distribuida se denomina PUMA®, que es desarrollado por Synopsys®, y es parte del pipeline de post-procesamiento de los MDP de circuitos, con el objetivo de reemplazar el software actual CATS®. PUMA® está en proceso de desarrollo, y uno de los objetivos de Synopsys® es que tenga un *performance* superior a CATS®, tenga más funcionalidades que éste, y pueda ser ejecutado en múltiples máquinas de manera sencilla.

PUMA® recibe como entrada un archivo de formato `tcl`, el cual describe los procesos a realizar, como también el archivo de entrada y el de salida. El archivo `tcl` es procesado y se genera un *OpGraph*, que es un grafo direccionado el cual representa el flujo de ejecución, con el respectivo procesamiento de los datos en cada nodo. Cada proceso puede ser ejecutado por una cantidad definida de *tasks* o tareas, las cuales hacen una pequeña parte del trabajo. Estas tareas necesitan sus propios datos de entrada y generan sus propios datos de salida, los cuales son leídos y escritos en archivos de datos, que posteriormente son leídos por el siguiente *task* que necesita ocuparlos.

El programa PUMA® puede ser ejecutado de dos maneras, en modo de ejecución local (modo sp), o en ejecución distribuida (modo dp). En el primer caso los *tasks* señalados

anteriormente, son ejecutados dentro de la misma máquina en la cual es ejecutado el programa. Para el segundo caso, los *tasks* se distribuyen entre una lista de máquinas que se provee a PUMA®, al iniciar su ejecución.

Uno de los objetivos específicos de esta Memoria, es poder probar el funcionamiento de un sistema de memoria distribuida. Para ello se toma la forma de escritura y lectura que se hace en PUMA® entre *tasks*, y se reemplaza por un sistema de memoria distribuida. En una primera instancia, se implementa para el modo de ejecución local, realizando pruebas para ver el rendimiento que tiene, y comparando con una versión sin escritura a memoria distribuida. En una segunda instancia se implementa la memoria distribuida para la ejecución en máquinas distribuidas, y se realiza nuevamente comparaciones con y sin los cambios.

Específicamente, los cambios se ejecutan en la escritura y lectura de datos entre *tasks*, que realizan una función llamada fractura. Esta función toma las figuras geométricas del plano MDP y las divide en sub-figuras más pequeñas y simples, que son rectángulos y trapezoides. Posteriormente, son escritas en el formato que las máquinas de fotolitografia ocuparán para crear los circuitos integrados. Los datos que se transmiten entre *tasks* son objetos definidos en PUMA®, que contienen estas figuras, que son guardadas en forma de archivos de datos binarios, y con los cambios serán guardadas en la memoria distribuida.

## 3.2. Implementación

La implementación de una memoria distribuida se realiza sobre PUMA®, que está actualmente en producción. Para ello se utiliza una librería desarrollada internamente en Synopsys®, llamada Common Data Store Library o CDSL. CDSL es una librería para aplicaciones distribuidas para guardar y compartir datos, usando una simple interfaz de llave-valor.

La librería CDSL está compuesta por dos interfaces de programación de aplicación (API) desarrolladas en el lenguaje de programación C. Una interfaz para el servidor, la cual controla y configura las máquinas que servirán como memoria distribuida, y la interfaz del cliente, la cual se ocupa en el programa que quiere acceder a los datos guardados.

El API del servidor contiene:

- Empezar, parar y configurar el servidor.
- Logear y archivar datos del servidor.

- Manejar el *pool* de servidores Redis que almacenan los datos (*backend*).

Las cuatro principales clases de comandos para la interfaz del cliente que posee CDSL, son:

- Llave-Valor: el cual obtiene, guarda, borra o verifica si existen los datos.
- Publicación/Suscripción: distribuye datos que mandan los publicadores a cada uno de sus suscriptores.
- Operaciones de conjuntos: inserción, búsqueda, remoción de elementos en un conjunto.
- Comandos de sesión: conectar, ping al servidor, guardar estado, estadísticas.

El trabajo realizado ocupa principalmente los comandos de llave-valor de la API del cliente.

Como PUMA® trabaja en base a distribución de pequeñas tareas o *tasks*, que se ejecutan en distintos *threads* del sistema, se aumenta el paralelismo en las tareas ejecutadas, y se puede aumentar el rendimiento en tareas de gran tamaño.

El principal problema que ésto conlleva, es la manera de pasar los datos de un *task* a otro. Por la naturaleza de éstos, tienen que ser independientes entre sí, ya que pueden estar corriendo en computadores totalmente diferentes. La manera de transferir información entre los *tasks*, es que éstos escriben a un archivo binario en disco duro, que será leído, posteriormente, por los siguientes *tasks*. Ésto genera que se lea y escriba a disco muchas veces en una ejecución del programa. Se ocupa una llave única para la identificación de los *tasks* y para los archivos de traspaso de información.

Con el objetivo de aumentar el rendimiento en esta área, se cambia la escritura a disco duro por escritura a la memoria distribuida proporcionada por CDSL, manteniendo el formato de llave única. Al ser guardado en la memoria distribuida, los datos quedan en la memoria RAM, siendo accesibles por todos los *tasks* independiente del computador en el que se encuentre el *task* e independiente de quien lo escribió.

Las modificaciones del código de escritura y lectura de los datos de entrada y salida de los *tasks*, consisten simplemente en cambiar de un llamado a escribir a un archivo binario con *path* único, a un llamado a escribir data binaria a CDSL con una llave única.

### 3.2.1. Secuencia de Datos o Streams

Un *stream* se entiende como una secuencia de datos que está disponible a lo largo del tiempo. Puede ser entendido como una cinta transportadora que permite a los elementos ser procesados uno a la vez, a diferencia de en grupos.

Como sistema de acceso a los servidores de CDSL, se crea una interfaz de *streams* como capa intermedia, de manera que para los componentes internos de PUMA® no sea necesario manejar la API de CDSL, dando una forma limpia y sencilla de escribir y leer diferentes datos a los servidores del sistema de memoria distribuida.

Se implementaron dos tipos diferentes de *streams*, uno de entrada o lectura y uno de salida o escritura. Para el de entrada se crea, en base a una llave dada previamente al constructor, que internamente llama a la API de CDSL para crear una conexión al servidor, que posteriormente se utiliza cuando se realiza una llamada de lectura. Los datos que retorna CDSL, los entrega el *stream* en forma de vector de caracteres.

Para el segundo caso, el *stream* de salida también recibe una llave en su constructor, y abre una conexión a los servidores de CDSL. Cada vez que algún dato es entregado al *stream*, éste los almacena dentro de un vector de caracteres, y solo cuando el stream es vaciado o destruido, éste realiza una llamada de escritura a CDSL.

La implementación de los *streams* se realiza para dar homogeneidad al acceso que se tiene a CDSL, como también mayor simplicidad a la hora de leer y escribir desde el sistema de memoria.

### 3.2.2. Atomicidad de Operaciones

Entre los requerimientos solicitados por parte de Synopsys®, se refiere a que los datos almacenados en CDSL no se corrompan, debido a que existe más de una instancia de PUMA® que está accediendo al mismo dato. Esto se puede deber a que dos instancias, que corren en paralelo, quieran escribir sobre el mismo registro y que la escritura de la segunda borre lo que escribió la primera instancia. Para evitar ésto, el sistema de memoria distribuido tiene que tener implementado un sistema de acción atómica, ésto quiere decir que mientras se lee y escribe en un registro, ni una u otra instancia con acceso a éste, tiene capacidad de realizar cualquier acción.

Con el requerimiento anterior en consideración, se efectúan pruebas en conjunto con las pruebas de rendimiento, en las cuales se asegura que la integridad de los datos no se

corrompiera. Para ello se agregaron llamadas de escritura y lectura adicionales, en las que todas las instancias de PUMA® (llegando a 128 en algunas pruebas) escriban y lean al mismo valor dentro de CDSL, sumándole uno para realizar un registro, y al final de la ejecución se compara el resultado con la cantidad de *tasks* que se generan por la prueba.

En todas las pruebas realizadas, se logra total consistencia entre el número de llamadas al valor de prueba, y la cantidad de tareas que se generan. Para el caso de un archivo de tamaño 70 kilobytes se generan 2.925 *tasks* diferentes, y se obtienen 5.851 `tasks` en la librería CDSL, que corresponde al doble de los *tasks* generados, ya que se lee y escribe una vez por cada uno, más uno que es el valor inicial de éste.

## 3.3. Implementación para Múltiples Máquinas

La implementación realizada anteriormente, se ejecuta en una versión de PUMA®, en la cual las tareas o *tasks* están dentro de una misma máquina. Durante el desarrollo del presente trabajo de Memoria, se le agregó una nueva funcionalidad a PUMA®, la cual permite ejecutar el programa en más de un computador (anteriormente solo se podía ejecutar en un solo computador). Específicamente, se puede ejecutar en distintas máquinas, teniendo que comunicarse los diferentes *tasks* por medio de la red, y haciendo más útil un sistema de memoria distribuido.

Los principales cambios realizados, se enfocan a soportar la nueva forma de ejecución de PUMA®. Principalmente la forma de ejecutar en múltiples máquinas o modo DP, debido a que la ejecución inicial, necesita pasar el parámetro de configuración de cuántas y cuáles máquinas dentro de la intranet se ocuparán, así como cuántas tareas cada máquina ejecutará al mismo tiempo.

Con estas modificaciones en consideración, se cambia el código previamente ocupado en la primera implementación, para adaptarlo solo al uso de PUMA® en múltiples máquinas, y también el uso del archivo de configuración dado, con las máquinas a ocupar. De esta forma, solo la ejecución de PUMA® en formato *master*, que es la instancia de PUMA® que controla todas las otras instancias (llamadas *workers*), inicializa el servidor de CDSL, dejando a los *workers* solo con acceso de lectura y escritura al sistema de memoria distribuido.

# Capítulo 4

# Discusión de Resultados

## 4.1. Comparación de Datos

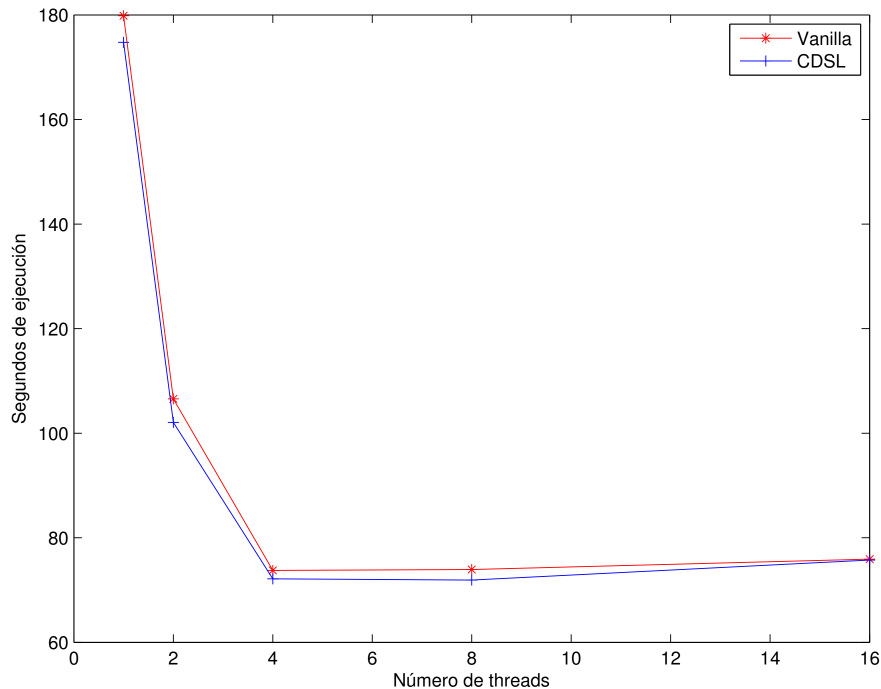
Para comparar el desempeño de cada programa, se realizan pruebas con el prototipo ya implementado, en los *clusters* habilitados por Synopsys®. De estas pruebas se obtienen los datos de rendimiento del prototipo, tanto de tiempo de ejecución, de demoras en la llamada al servidor DSM y de precisión de los resultados. Los datos obtenidos se comparan con las mismas pruebas, pero realizadas con la versión anterior del programa que se tenía en uso. De esta forma se puede comparar gráficamente el desempeño de cada programa.

## 4.2. Resultados Preliminares

Los cambios realizados (modificar la manera de almacenar los datos entre un *task* y otro), entregan una mejora marginal con respecto a la versión sin modificar (ver figura 4.1). El experimento se realiza sobre una misma máquina, sin tener la memoria distribuida sobre una intranet.

Si bien el cambio entrega una mejora marginal, en las condiciones que se desarrolla el experimento, con mucha variabilidad de los resultados, el comportamiento es el mismo que el original al ir aumentado la cantidad de *threads*. Sin embargo al eliminar el cuello de botella que es leer a disco, se espera tener mejores resultados al escalar el experimento, como es el incremento del tamaño de las pruebas y el número de computadores involucrados.

Figura 4.1: Resultados preliminares con CDSL



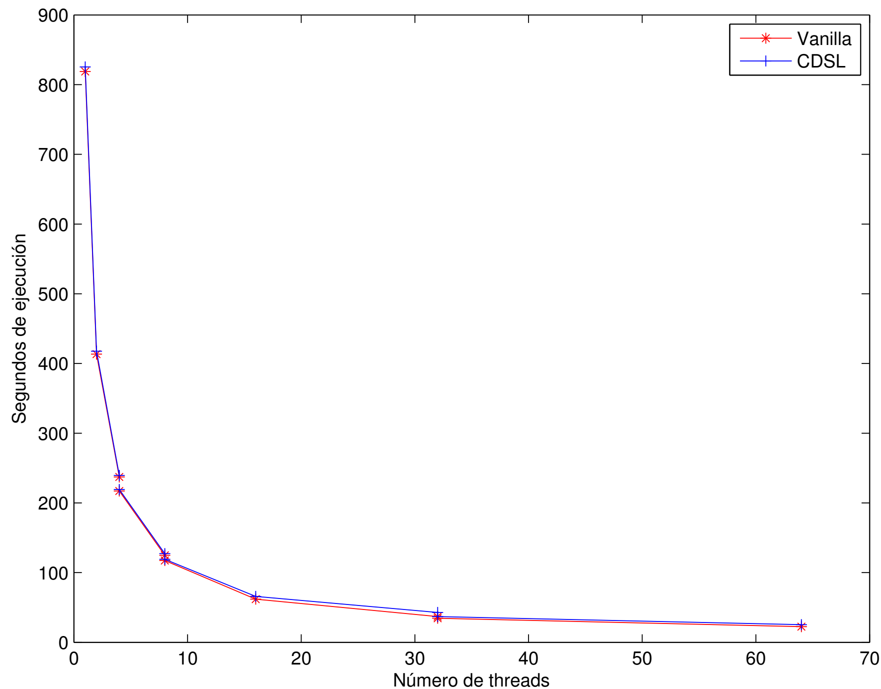

## 4.3. Resultados

Los resultados del primer conjunto de pruebas realizadas, se desarrolla sobre la modificación del código, que incorpora CDSL, el cual puede ejecutarse en diferentes *tasks* y computadores.

En estas pruebas se ocupa un archivo de tamaño pequeño (70 kilobytes de entrada y 8.7 megabytes de salida), con el cual se obtuvieron resultados negativos. Los tiempos de ejecución en todos los casos, fueron más lentos que en la situación que no se incluye CDSL a PUMA®.

Ésto se debe a que la reducción del tiempo en el procesamiento de los datos, no es suficiente para compensar el *overhead* (tiempo adicional) que se utiliza para inicializar los servicios del lado del servidor que ocupa CDSL. Por lo tanto, aunque el procesamiento en sí del archivo es menor que sin CDSL, con el *overhead* generado, el tiempo total de ejecución es mayor con la modificación implementada.

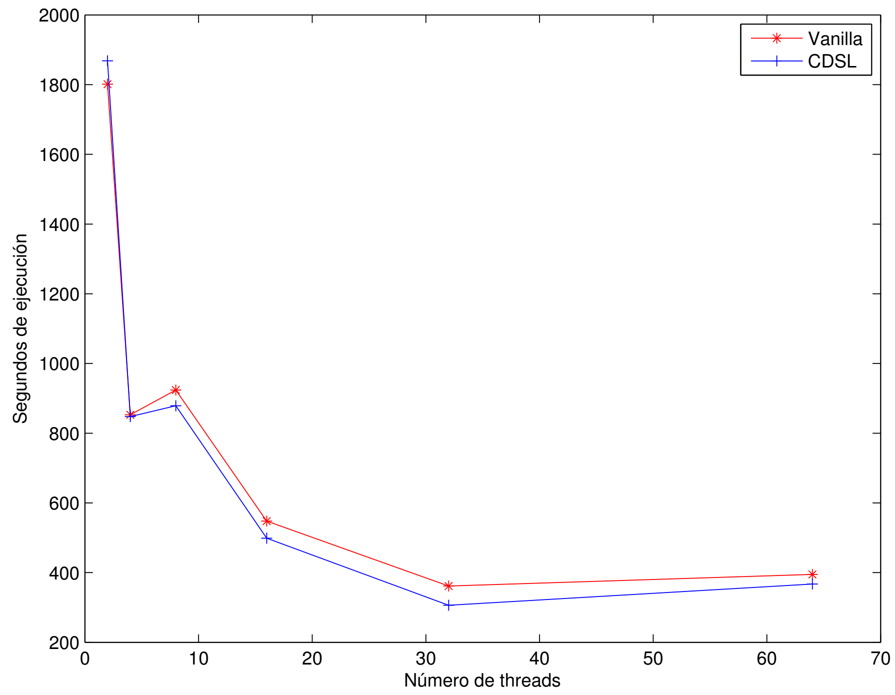
Figura 4.2: Resultados preliminares de `dp` con CDSL archivo de entrada 70 kilobytes




En un segundo conjunto de pruebas realizadas, en donde se ocupa un archivo de prueba con un tamaño de entrada aproximado de 468 megabytes, el cual después del procesamiento queda con un tamaño de 222 megabytes. En la figura 4.3, se pueden ver los resultados obtenidos con este archivo, y la mejora significativa en el tiempo de ejecución que se obtiene a lo largo de distintas configuraciones de *tasks*. Se observa que al aumentar los *tasks*, la implementación con CDSL mejora sus tiempos de ejecución, llegando a más de un 10 % de mejora en relación a la implementación sin CDSL, en el tiempo de ejecución.

En los gráficos generados de las pruebas realizadas, se puede apreciar el efecto que tienen en primer lugar, la cantidad de *threads* que se ocupan para correr la solución, que disminuyen el tiempo de ejecución en todos los casos. En segundo lugar, la diferencia en rendimiento dependiendo del tamaño de los archivos de entrada que se ocupan, obteniendo mejores niveles de rendimiento a mayor tamaño.

Figura 4.3: Resultados en dp con CDSL archivo de entrada 468 megabytes



# Capítulo 5

# Conclusiones

Del trabajo realizado en esta Memoria, se obtuvieron las conclusiones que se presentan a continuación.

La primera conclusión es que se logra analizar diferentes tecnologías que resuelven el problema inicialmente planteado, con los requerimientos impuestos desde el punto de vista de la construcción interna del software PUMA®, y por Synopsys®. Se logra encontrar una tecnología que se adapta mejor a las características del problema. Entre las alternativas probadas, se escogió CDSL, que resulta ser una buena elección por sobre las alternativas open-source, ya que al ser un producto desarrollado internamente por Synopsys®, tiene una mejor integración con los otros productos desarrollados, como PUMA®. A diferencia de una solución externa, que tendría que haber sido adaptada y agregada al software que lo requería.

Como segunda conclusión, se mejora el tiempo de ejecución en algunos de los archivos y configuraciones probadas; tomando en cuenta el tamaño de los archivos ocupados que alcanzan los cientos de gigabytes en algunos casos, y son procesados en diferentes máquinas. Ésto muestra que un sistema de memoria distribuida, es una buena forma de abordar el problema de leer un mismo conjunto de datos desde múltiples máquinas, y que además funciona más rápido que un disco duro.

Dentro de los resultados obtenidos, el aumento del número de *threads* en la ejecución de PUMA®, mejora los tiempos de ejecución. Pero se aprecia que la ganancia por cada nuevo *thread* que se incorpora es cada vez menor, generando una curva asintótica en el gráfico, tendiendo al tiempo mínimo de ejecución. Se puede concluir para este caso que hay un número ideal de *threads* (dependiendo del archivo de entrada), que equilibra la ganancia en el rendimiento de PUMA®, con el costo de ocupar varias máquinas en la ejecución de

PUMA®.

También es importante señalar, que se presentan resultados negativos al procesar el primer conjunto de datos, en el cual se utiliza un archivo de entrada pequeño, puesto que la solución ocupada está orientada, como se señala en el punto anterior, a grandes volúmenes de datos. Se puede concluir que el éxito de un sistema de memoria distribuida, depende del tipo de dato de entrada que se ocupe. Si los datos de entrada no son capaces de compensar el *overhead* que se produce al introducir el sistema de memoria, no es beneficiosa la implementación de dicho sistema de memoria distribuida.

Otro aspecto relevante de señalar, es la capacidad de retención de datos del sistema de memoria distribuida. Si bien escribir los datos a disco es un proceso más lento, si llegase a tener un error la máquina, o se apaga súbitamente, los datos del procesamiento son posibles de recuperar. En cambio, en el sistema de memoria distribuida al momento de fallar la máquina, todos los datos hasta ese entonces procesados se pierden, ya que la memoria RAM se borra. Esto puede ser evitado parcialmente, si se implementan sistemas de respaldo, de manera que cada cierto tiempo se escribe todo lo que está en RAM a un disco duro. Pero solo se recupera hasta el último punto de respaldo, y no al instante de la falla.

Finalmente y dado que todo el trabajo se desarrolla en la empresa Synopsys®, ocupando su infraestructura, es importante destacar que se logra implementar toda esta Memoria, dentro de un software de escala empresarial como es PUMA®, con toda la complejidad que ello involucra. Se espera que con los resultados de este trabajo, se ayude a mejorar el rendimiento del software en el procesamiento de datos, aumentando la rapidez con que se procesan las máscaras de los circuitos integrados. La implementación de este cambio permite que Synopsys® mejore el rendimiento de su trabajo y consecuentemente el de sus clientes.

# Bibliografía


[1] *Redis data store*. `http://redis.io`.

[2] Baker, R. Jacob: *CMOS: Circuit Design, Layout, and Simulation*. Wiley-IEEE, tercera edición, 2010.

[3] Fitzpatrick, Brad: *Distributed caching with memcached*. Linux Journal, (124), agosto 2004.

[4] Lavagno, Martin y Scheffer: *Electronic Design Automation For Integrated Circuits Handbook*. 2009.

[5] Patterson, David A. y John L. Hennessy: *Computer architecture : a quantitative approach*. Morgan Kaufmann Publishers, cuarta edición, 2007.

[6] Zawodny, Jeremy: *Redis: Lightweight key/value Store That Goes the Extra Mile*. Linux Magazine, agosto 2009.

# Apéndice A

Datos de los resultados de las pruebas realizadas.

| tasks | tiempo total acumulado | tiempo task master (user/system) |
|---|---|---|
| 4 pc/4 task | 1801.183706s | 4.750000s / 0.410000s |
| 2 pc/8 task | 852.598074s | 5.260000s / 0.430000s |
| 4 pc/8 task | 923.776283s | 21.820000s / 1.470000s |
| 4 pc/16 task | 547.933851s | 5.730000s / 1.120000s |
| 4 pc/32 task | 361.643598s | 22.930000s / 1.330000s |
| 4 pc/64 task | 394.717826s | 25.270000s / 2.210000s |

Tabla A.1: Datos de archivo de entrada grande sin CDSL en múltiples máquinas

| tasks | tiempo total acumulado | tiempo task master (user/system) |
|---|---|---|
| 4 pc/4 task | 1868.164940s | 4.810000s / 0.340000s |
| 2 pc/8 task | 847.370219s | 5.140000s / 0.370000s |
| 4 pc/8 task | 878.484107s | 5.450000s / 0.440000s |
| 4 pc/16 task | 498.842636s | 21.780000s / 1.070000s |
| 4 pc/32 task | 306.165661s | 22.950000s / 1.120000s |
| 4 pc/64 task | 367.166420s | 9.150000s / 1.120000s |

Tabla A.2: Datos de archivo de entrada grande con CDSL en múltiples máquinas

| tasks | tiempo total acumulado | tiempo task master (user/system) |
|---|---|---|
| 1 pc/1 task | 818.871277s | 0.410000s / 0.110000s |
| 1 pc/2 task | 413.456286s | 0.490000s / 0.110000s |
| 1 pc/4 task | 237.149638s | 0.620000s / 0.130000s |
| 2 pc/4 task | 216.950043s | 0.620000s / 0.120000s |
| 2 pc/8 task | 124.766476s | 0.930000s / 0.160000s |
| 4 pc/8 task | 117.271077s | 0.920000s / 0.140000s |
| 4 pc/16 task | 61.968518s | 1.510000s / 0.200000s |
| 4 pc/32 task | 36.982992s | 2.660000s / 0.360000s |
| 8 pc/32 task | 34.427243s | 2.630000s / 0.340000s |
| 8 pc/64 task | 22.368229s | 4.910000s / 0.630000s |

Tabla A.3: Datos de archivo de entrada pequeño sin CDSL en múltiples máquinas

| tasks | tiempo total acumulado | tiempo task master (user/system) |
|---|---|---|
| 1 pc/1 task | 825.363044s | 0.480000s / 0.110000s |
| 1 pc/2 task | 417.597427s | 0.550000s / 0.130000s |
| 1 pc/4 task | 239.391900s | 0.680000s / 0.150000s |
| 2 pc/4 task | 219.318509s | 0.680000s / 0.140000s |
| 2 pc/8 task | 127.302931s | 0.980000s / 0.160000s |
| 4 pc/8 task | 119.149760s | 0.980000s / 0.170000s |
| 4 pc/16 task | 65.884726s | 1.540000s / 0.220000s |
| 4 pc/32 task | 42.778593s | 2.720000s / 0.360000s |
| 8 pc/32 task | 36.925877s | 2.660000s / 0.340000s |
| 8 pc/64 task | 25.299657s | 5.050000s / 0.630000s |

Tabla A.4: Datos de archivo de entrada pequeño con CDSL en múltiples máquinas

| threads | tiempo total acumulado | tiempo task master (user/system) |
|---|---|---|
| 16 `threads` | 75.926688s | 306.120000s / 644.610000s |
| 8 `threads` | 73.914965s | 283.820000s / 187.710000s |
| 6 `threads` | 74.106185s | 240.340000s / 114.710000s |
| 5 `threads` | 70.762320s | 210.630000s / 69.730000s |
| 4 `threads` | 73.754432s | 200.170000s / 39.900000s |
| 3 `threads` | 81.539505s | 188.430000s / 19.510000s |
| 2 `threads` | 106.564958s | 176.160000s / 11.770000s |
| 1 `thread` | 179.846285s | 154.540000s / 4.010000s |

Tabla A.5: Datos de archivo de entrada pequeño sin CDSL en una sola máquina

| threads | tiempo total acumulado | tiempo task master (user/system) |
|---|---|---|
| 16 `threads` | 75.727579s | 321.370000s / 692.650000s |
| 8 `threads` | 86.853413s | 280.670000s / 209.060000s |
| 4 `threads` | 72.131227s | 206.730000s / 37.960000s |
| 2 `threads` | 102.057163s | 174.180000s / 9.800000s |
| 1 `thread` | 174.760767s | 157.350000s / 4.460000s |

Tabla A.6: Datos de archivo de entrada pequeño con CDSL en una sola máquina