ForPowER: A Novel Architecture for Energy Efficient Implementation of Fork-Join Parallelism Using System on a Chip

A Thesis

Submitted to the Faculty

of

Drexel University

by

Zayd Hammoudeh

in partial fulfillment of the

requirements for the degree

of

Master of Science in Computer Engineering

May 2006

TABLE OF CONTENTS

List of Tables

List of Figures

Abstract

ForPowER: A Novel Architecture for Energy Efficient Implementation
of Fork-Join Parallelism Using System on a Chip
Zayd Hammoudeh
Nagarajan Kandasamy, Ph.D.
Moshe Kam, Ph.D.

We describe ForPowER, a power-efficient architecture for handling fork-join parallelism using system on a chip. Our design consists of 16 processor cores, capable of dynamically scaling their clock frequencies and supply voltages under different workloads. The processors are divided into four sets of four, with each set sharing a multiported two-level cache. This arrangement reduces the energy wasted on powering redundant data. ForPowER also uses a central scheduler, which assigns tasks to the processors, taking advantage of the shared memory and of the processors ability to scale their clock frequencies under varied workload.

We also describe power models for all components of the SoC design, namely the caches, processors, and the network.

We show that in simulation, ForPowER outperforms the most widely used fork-join architecture on the SPEC-95 Hydro2D benchmark, consuming over 65% less energy.

Chapter 1 - Introduction

Since the early 1990's, there has been explosive growth in the performance and capabilities of computer systems, with an increase in speed of over 150 times [28]. One of the catalysts for this has been the exploitation of parallelism in software [1]. Parallelism is the level at which multiple instructions in a section of code can be issued and performed concurrently. In traditional serial computing, each task must be executed sequentially on a single processing element. In contrast, a parallel computer is "a collection of processing elements that communicate and cooperate to solve large problems fast" by distributing those portions of a program that can be parallelized to be executed simultaneously on different processors. Extracting the benefits of parallelism involves incorporating additional hardware into computer systems. As opposed to achieving improved performance via adding supplementary functional units (e.g. additional pipelines, ALUs, floating point units) to a single processor, the most common approach to parallel computing is to link multiple processing units together using a communication architecture. This prevents the processors from becoming unnecessarily complex by reducing the total number of functional units on a single processor by using multiple, connected processors.

This thesis will focus on one specific type of software parallelism known as fork-join parallelism. Fig. 1.1 shows the execution-flow of a program that fits fork-join parallelism, or can be made to fit that paradigm. Following a section of a program where instructions are performed sequentially, the subsequent execution forks into R independent jobs. Each job is comprised of a set of tasks that must be executed sequentially. For a given job k, there are r_k tasks. While each job's tasks need to be

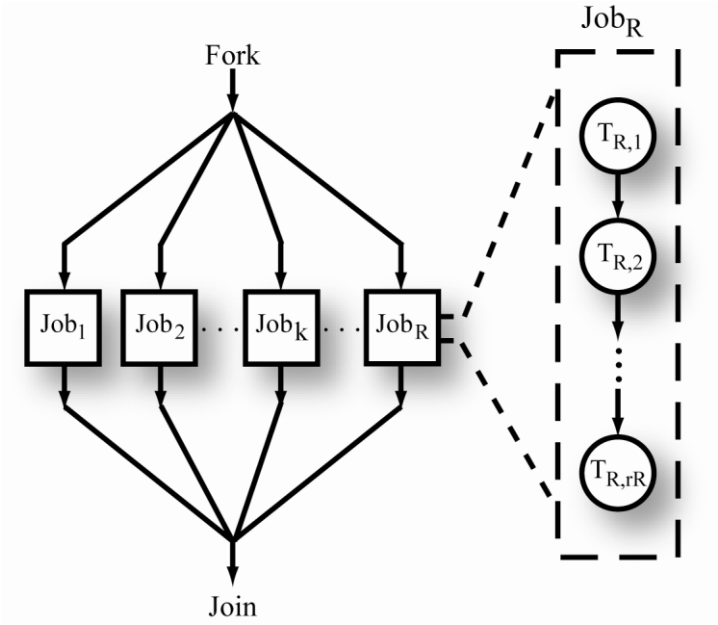

Figure 1.1 - Fork-join program model with *R* jobs. Each job *k*, comprises a set of tasks ($T_{k,l}$) that are executed sequentially

Executed sequentially, the *R* jobs can be scheduled to different processing units to execute in parallel. Task *l* within job *k* has an expected execution time. As such, the total execution time of job *k* is the sum of the execution times of all r_k tasks. When all *R* jobs have completed execution, they synchronize, which entails waiting for the remaining jobs to complete and performing any necessary merging of their outputs. Once synchronization completes, sequential execution can continue.

Previous attempts to design parallel computers aimed at fork-join programs have expanded on work performed by Kumar in [3]. It has been used as the basis for other work in the area, including [4] and [30].

Kumar's architecture, shown in Fig. 1.2, consists of a central scheduler that functions as a simple Round Robin scheduler [33]. Upon arrival of a job at the scheduler,

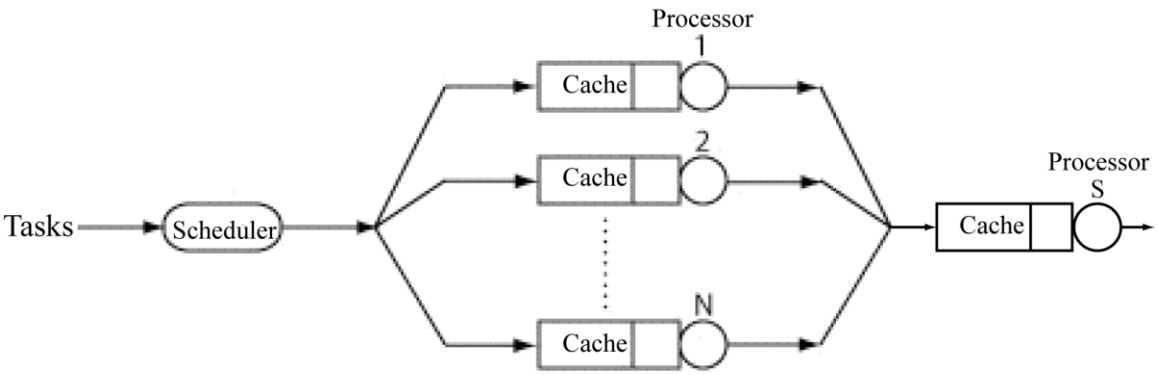

Figure 1.2 - Model of Kumar's architecture for handling fork-join parallelism, which links a central scheduler to N processors, with two level caches. These then are connected to a synchronization processor, S.

the job is sent over a point-to-point network [33] to one of *N* homogeneous (identical) processors in the system. Each processor has its own dedicated, two-level cache that stores the data sent by the scheduler and any variables created during execution. Upon completion of all the jobs in a fork, the processors send their data to the synchronization processor *S*, which synchronizes all of the jobs.

Despite its use, Kumar's architecture suffers from serious limitations. The most notable among them is that the architecture fails to take advantage of the fact that not all jobs in a given fork have the same execution time. In a fork, the set of jobs that has the largest execution time constitute the critical job. The critical job provides a lower bound for the execution time of the entire fork; this arises from the fact that before synchronization can occur, all other jobs must wait for the critical job to complete. By reducing the execution time of the critical job, the execution time for the entire fork is reduced.

Since those jobs that are not the critical job must wait for the critical job to complete before synchronization can occur, a time slack exists in such jobs. This allows

for the overall execution time of those jobs that are not critical to be delayed until their execution time equals the execution time of the critical job, without impacting overall system performance. When the jobs are assigned to different processors to be executed, the critical job should always be run at the maximum operating frequency, because it is the one requiring the most time to complete. An effective way to slow down the execution of non-critical jobs is to assign them to processors with clock speeds less than the processor executing the critical job (this technology was not available to Kumar when he developed his architecture). Work in dynamic prediction of expected execution time for a given section of a program, which allows for the adaptive scheduling of jobs, has been done by Gergeleit [5].

Fig. 1.3 is a demonstrative example of a fork-join with the format of this figure

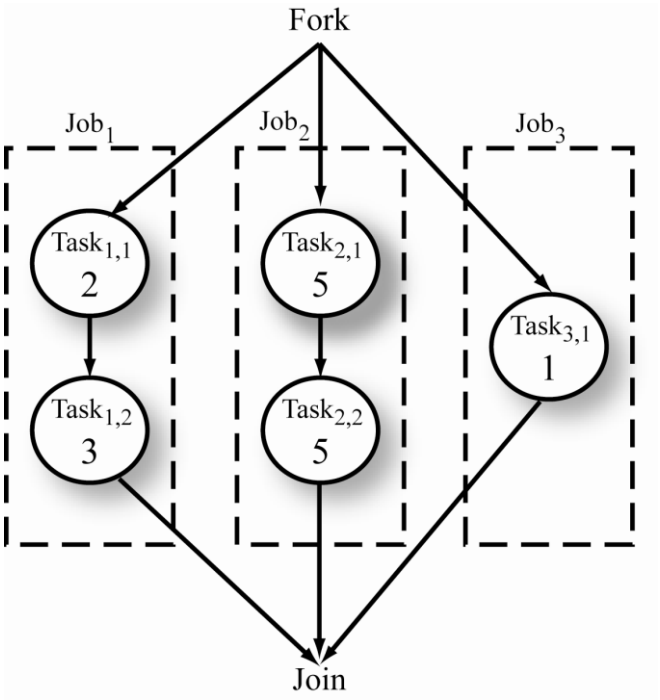

Figure 1.3 - Sample fork and join. The execution time of each task is indicated below the task name.

similar to that found in Fig. 1.1; the circles contain the name of each task and its corresponding execution time. Since the total execution time of a job is the sum of the execution times of all the tasks that constitute it, job_2 (i.e., $Task_{2,1}$ and $Task_{2,2}$) is the critical job with a total time of 10 compared to 5 for the job_1 ($Task_{1,1}$ and $Task_{1,2}$) and 1 for the job_3 (task). Those tasks not part of the critical job (e.g. $Task_{1,1}$, $Task_{1,2}$, and $Task_{3,1}$) can be scheduled to slower processors without affecting the net time to complete the fork. Since the sum of the execution times of $Task_{1,1}$ and $Task_{1,2}$ is half that of the critical job, the processor that is executing those tasks can run at half the speed as the one executing the critical job (i.e., $Task_{2,1}$ and $Task_{2,2}$), as shown in [29] that execution time scales linearly with processor speed for tasks that are executed on the CPU. Similarly, the processor executing $Task_{3,1}$ can run at one-tenth the speed of the critical job. This allows for near simultaneous synchronization, which has the distinct advantage that by slowing the processor speed of non-critical jobs, significantly less power will be consumed than by running at faster speed and idling the processor [18].

Moreover, each processor executes its required workload independent of the other processors. However, Martorell in [7] suggested grouping different processors into different sets of processors. Each set of processors consisted of a master processor and possibly several slave processors; the master and slaves processors had different roles with the master processor starting executing and then spawning, if necessary, any parallelism encountered to the slave processors. Through this grouping arrangement, Martorell notes that the processors in a group are able to cooperate with each other to complete execution of encountered parallelism more efficiently. Similarly, Martorell

notes that by grouping processors into smaller groups, it then makes it worthwhile to attempt to exploit multiple levels of parallelism that can be present in fork-join programs.

Another limitation of Kumar's architecture is that each of the N processors is assigned a dedicated cache. The authors of [6] note that the jobs in a fork all share a portion of data and instructions. As such, it is inefficient to give each individual processor a dedicated cache as it takes additional network resources to resend the shared data to each of the processors, as well as expend energy preserving multiple copies of this shared data in these caches.

In addition to the power saved by using shared memory, others have studied the performance and implementation benefits of shared memory for fork-join programs. Quinn in [6] noted that by allowing for shared memories between the different jobs in the fork, the synchronization processor can use the shared program variables that are stored in these shared memories, which drastically simplifies synchronization. Similarly, if one job in a fork needs to send it a message to another job, it can do so by writing to a shared memory location. In contrast, Kumar's architecture used a message passing strategy, which is a form of communication where messages are sent through the network to communicate with the different jobs. As such, when synchronization occurs, the different processors must send the data through the network and create new variables at the synchronization processor. The sending of the requisite data and creating the new variables is additional overhead added by this communication strategy.

Furthermore, the architecture proposed by Kumar in [3] did not take advantage of a new and increasingly popular method for linking multiple processing elements known as System on a Chip (SoC). An SoC design is an application specific integrated circuit

(ASIC) which consists of different (i.e. heterogeneous) types of components on a single chip that communicate with each other and work together. Potential SoC nodes include: memories, I/O devices, digital signal processors (DSPs), and central processing units (CPUs). These components are linked through a system of interconnects between the different components. Types of interconnects include: a shared medium (e.g. bus) and a point-to-point network. Through this collection of diverse nodes, an SoC serves as a complete computing system that can execute software or programs stored in on-chip memories or inputs from the I/O ports.

Compared to computer systems, such as Kumar's [3], where the nodes are not all connected to each other on a single chip, system on a chip has several distinct advantages for a small number of nodes (typically under one hundred). SoCs are designed with the capability to provide error-free communication between nodes [21]; this is possible because of the fact that all nodes are connected on the same chip with reliable communication channels. This ability leads the designers of SoCs to assume a fully deterministic interconnect architecture for communication (i.e., all messages are guaranteed to be delivered) as well as value resolution (the intended message will be correctly extracted from the received data), significantly simplifying the architecture for a low number of nodes.

Fig. 1.4 is a schematic of a multiprocessor SoC. The model has heterogeneous (i.e. nonuniform) placement of memory units as well as heterogeneous processing units. Since the architecture of SoCs is tuned and specialized to function optimally for a specific application or restricted set of related applications, selection of components and the design of the chips are nonstandard.

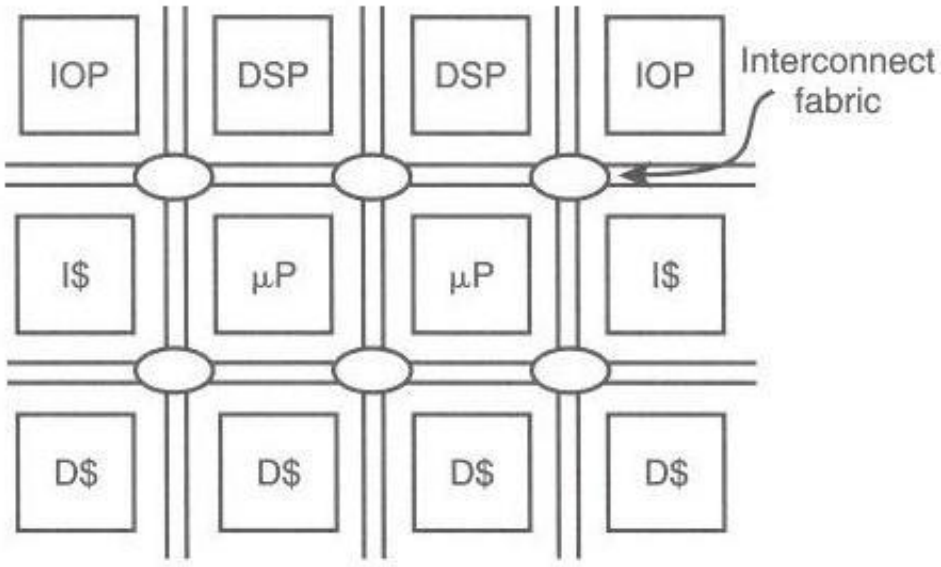

Figure 1.4 - Schematic of a modern SoC with DSP processors (DSP), microprocessors (µP), input-output ports (IOP), data caches (D$), and instruction caches (I$) connected with an interconnection fabric.

The overall performance of an SoC is based on three criteria: latency of execution, area efficiency, which is the size of the chip required for all the hardware, and energy efficiency, which is the energy consumed by the chip during execution. In embedded systems and portable devices, energy efficiency in SoCs is becoming a key design constraint. Ideally, it is important to limit the overall power consumption of the system without affecting the overall execution time of programs [2].

When designing a low-power computer architecture for a system on a chip, it is insufficient to solely focus on minimizing the energy dissipated by the processing units. The power dissipated by the interconnection networks linking the nodes together can be substantial; in the MIT RAW Network on a Chip Architecture, 36% of overall chip power was dissipated in the interconnects [31].

This thesis develops ForPowER, the Fork-Join Power Efficient aRchitecture. It is an energy efficient SoC design to exploit fork-join parallelism of the type shown in Fig. 1.1. By simulating the execution of the SPEC-95 Hydro2D benchmark that solves the hydronamical Navier-Stokes equations, we compare the energy consumed by ForPowER with Kumar's architecture shown in Fig. 1.2. The proposed ForPowER architecture is shown to consume 65% less energy than Kumar's design. We accomplish this by allowing ForPowER to dynamically slow down processor speeds of non-critical jobs allowing them to save significant portions of the energy. The power savings are further improved by permitting the sharing of data between the processors significantly, which limits the power wasted on redundant data.

The rest of this thesis is as follows. Chapter 2 describes ForPowER, our architecture for efficiently handling fork-join parallelism. Chapter 3 details ForPowER's scheduler and assignment algorithm for distributing jobs in the SoC. Chapter 4 presents our experimental results and ForPowER's load sensitivities. Chapter 5 discusses our conclusions and suggestions for future work.

Chapter 2 - Power Efficient Computer Architecture

The chapter will discuss ForPowER, our energy-efficient SoC for handling fork-join parallelism. Specifically, we will describe the caches, processors, and network topology that are being used. We also explain how power consumption is modeled in all three parts of the SoC.

Since fork-join programs only require a small number of processing elements (as was explained in [7] that the number of jobs in fork join programs does not usually exceed 64), the main constraint of scalability in SoCs is not a factor. The architecture discussed in this thesis selected 16 processors, as this is the standard size used in other studies such as [7]. All derived formulas and requisite hardware was designed based on 16 processors but is easily expandable to 32 or 64 nodes.

Fig. 2.1 is the proposed layout of ForPowER. ForPowER consists of multiple core sets (which are made up of 4 processors that share a single cache), an interconnection network, and a scheduler. These components will be designed to reduce the energy consumed.

We propose the use of four identical core sets. Each core set consists of four processors sharing a unified (i.e., it contains both instructions and data) multiported, two-level, banked cache [32]. The cache can be accessed simultaneously by all the processors. Moreover, all processors can be run at different clock frequencies (and in turn different supply voltages from the power source to the processor) to limit the power dissipated by the system.

The interconnection fabric is a router based, point-to-point network. All four core sets and the scheduler have a dedicated network interface. Each of the routers at the

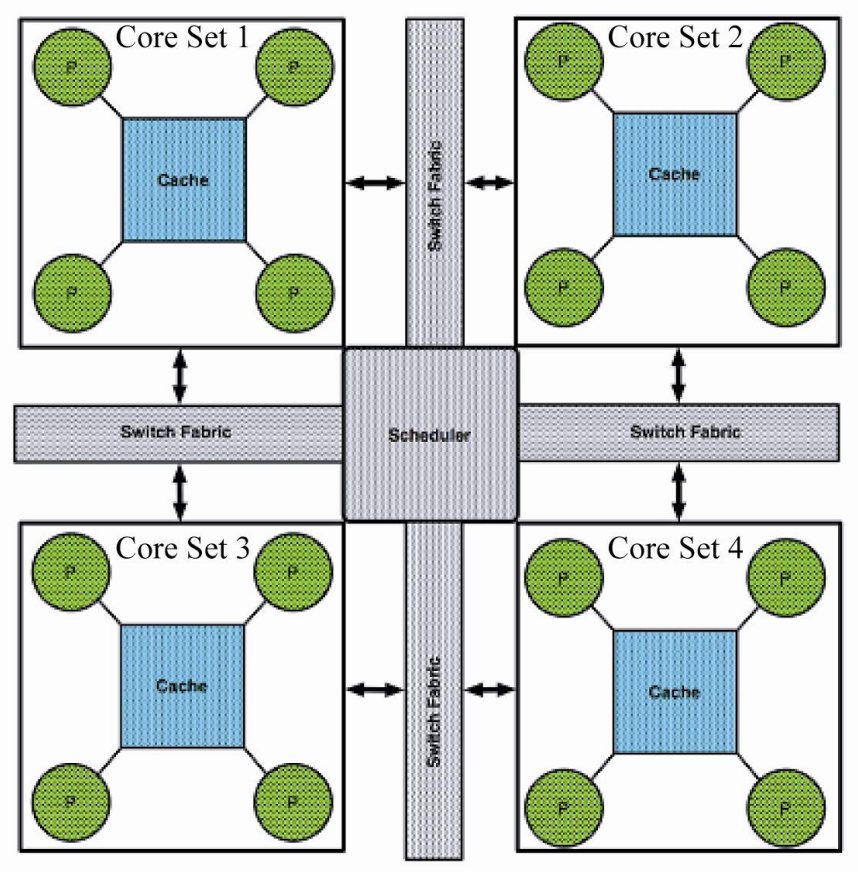

Figure 2.1 - The SoC layout of ForPowER. It consists of 16 processors (P), 4 multi-ported caches, a central scheduler, and a switch fabric. Each set of 4 processors and shared cache forms a core set.

interfaces is linked to all other routers, making the network fully connected at the router level.

The scheduler is designed as the central unit that assigns each task to a processor in the system, schedules synchronization, and sets the processor speeds. The scheduler uses an energy-efficient cost-based algorithm for task assignment that is detailed in Chapter 3.

2.1 Core Set Structure

The core set consists of two distinct components, the multiported cache and the four processors. Both are designed to limit power consumption.

2.1.1 Cache Architecture

The assumption throughout the literature is that all jobs created by a fork rely on some core of data that all the jobs use, and that data is left unchanged. Since each task operates independently only on the data it needs, no cache coherence protocol [1] to maintain integrity of the shared data is required, as data shared between jobs are never modified. In addition, rather than giving each processor a private cache, which would each be sent duplicate copies of the data, it is more energy efficient for multiple processors to be linked to a shared cache that contains only one copy of this shared data set. By using a shared cache, energy is saved by (1) eliminating the need to power identical blocks across multiple caches, (2) reducing network traffic by decreasing the number of copies of the data that need to be transmitted, and (3) limiting or eliminating the need to send all data to a single processor (e.g. the *S* processor in Kumar's architecture) when synchronizing. Note that if all the jobs are run off the same shared cache, then all the data is already present for any single processor to operate.

There are multiple methods to implement a shared cache. The simplest method is to share a single-ported cache amongst multiple processors. Although implementing a

shared single-ported cache requires less board area and is less complex to implement than a multiported cache, it causes significant degradation of performance and increases the execution time of parallel programs. By increasing the number of ports on a data cache to five (one for each of the four processors and the network interface), it has been shown in [9] that a 24.7% increase in performance can be obtained compared to using a single port. Therefore, a multiported cache was selected as it allows all processors and the network to access the cache simultaneously.

Multiported caches are of three distinct types: ideal multiporting, time-division multiplexing, and multiple independently addressable banks [10]. Ideal multiplexing allows each cache block to be simultaneously accessed by all cache ports. This structure, however, is impractical and never applied in industry because of the costs in area, power consumption, and access time [10].

Time division multiplexing type multiported caches utilize time to achieve virtual ports in a cache. Memory is accessed twice in each cache clock cycle (on both the rising and falling edges) allowing it to operate at twice the processor speed. However, this scheme is impractical because of its lack of scalability to multiple ports [10]. Its impracticality is exacerbated in ForPowER since processor cores are running at different speeds, making implementation of the protocol overly complex.

Given the impracticalities of the other cache organizations, the best solution for ForPowER is to use a recently developed technique that was not available to Kumar known as multi-banking to achieve multiple ported memories. Multibanking involves dividing the memory into multiple banks, the number of which is defined by the

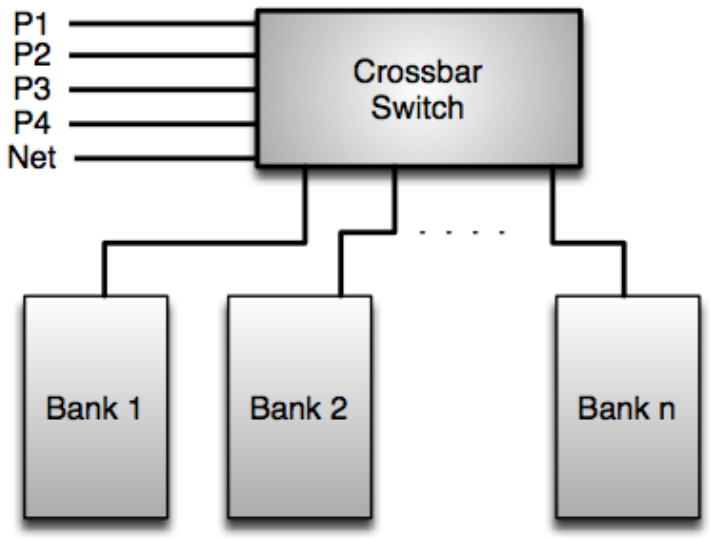

Figure 2.2 - Multiported cache architecture, with *n* banks

application space. Each bank is in essence a single-ported cache capable of handling a single access per cycle. The processors or the network access the individual caches through the use of a crossbar, which is a switch that connects the four processors and the network to the cache banks, as described in [11]. Fig. 2.2 shows the overall structure of the shared caches; it consists of *n* banks that are linked to the four processors and the network through the crossbar.

As previously noted, when multiple nodes try to access the cache, it must first go through the crossbar switch to determine the appropriate bank to connect to at which point it can then extract the data from that bank. As such, cache accesses require two clock cycles to complete; the first cycle is to use the crossbar to decode the proper bank while the second is to retrieve the data from the bank. If multiple nodes try to access the same bank in the same clock cycle, one of the nodes is granted access to the bank, while

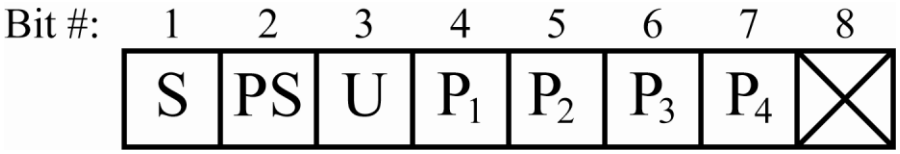

Figure 2.3 - Flag bits representing the state of a bank in the multiported cache. S and U represent if the bank is shared or unused respectively. If the bank is processor specific (PS), it can be set to be specific to any of the four processors in the core set (e.g., P_1, P_2, P_3, P_4).

the remaining nodes wait till that memory operation completes and then attempt to access the bank again.

To determine how the multiple nodes access the different cache banks, the banks can be in one of three states: shared, processor specific, or unused. How to divide the data between the banks and what its state should be when data is loaded is determined at compile time. The state of the caches is changed dynamically by the processor or the network and stored in specific memory locations in each bank that are eight bits long. Fig. 2.3 shows the bit association to the cache banks' states. Bits one through three are active-high flags for whether that bank is either shared, processor specific or unused respectively. Next, given a bank is processor specific (i.e., used by only one processor), bits four through seven are flag bits that signify which of the four processors in the core set the bank is associated with. Bit eight is not used.

Shared cache banks contain data used by more than one processor in the core set. For instance, if all the processors in the core set rely on shared data, it can be stored in a shared bank and accessed by all nodes.

Processor specific banks are used to create cache locations that can only be accessed by one specific CPU. This eliminates contention on memory operations on data that are needed by only one task.

Unused banks are those that contain no useful data. This occurs when no data have been stored in a location yet or if the data in that bank are no longer needed.

Fig. 2.4 is a simplified state diagram describing how a cache block transitions between states with transition abbreviations. All blocks are initially unused. Unused blocks can transition into the shared or processor specific states depending on if the data loaded is shared (DLS) or if the data loaded is processor specific (DLPS). Once a bank is

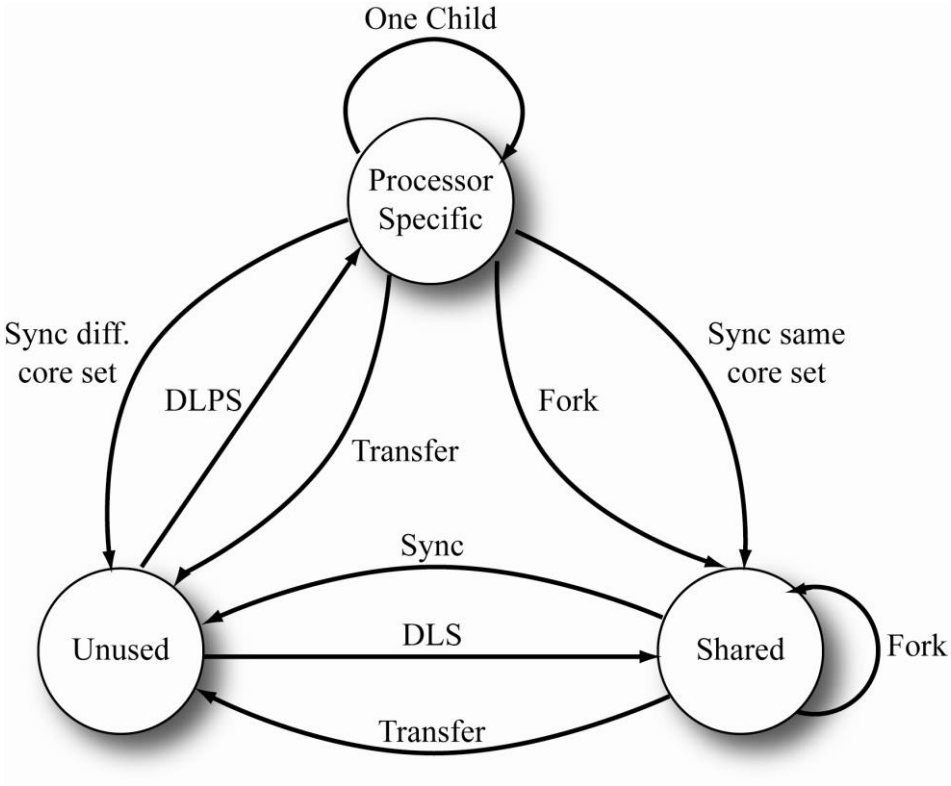

Figure 2.4 - Cache bank state transition diagram. Transitions between bank states occur when there is: a fork, synchronization (Sync), transfer of data to a different core set, processor specific data is loaded (DLPS) into the bank, or shared data is loaded into the bank (DLS).

in no longer unused, the events that can cause a transition between states include: a fork (F), a task which once completed only leads to one child (OC) task that can then execute, a synchronization (SYNC), and transfer (T) to a different core set.

When a bank is in the shared state, if the sequential execution forks or if a job finishes executing and is ready to synchronize, then the data should remain in the shared state until the synchronization has completed.

A bank in the processor specific state should remain in that state if it only has one child task (i.e., it is not a fork or synchronization) because it is still only needed by that processor. However, if a synchronization of a job on a specific processor is reached and synchronization for that processor will be completed on its core set, as determined by the central scheduler shown in Fig. 2.1, the bank associated with that processor should switch from the processor specific to the shared state. On the contrary, if synchronization is to occur on another core set, the data should be sent across the network.

Once all the data in a cache bank is no longer needed because synchronization has completed or the task has been transferred out of the core set, then the cache returns to the unused state.

Both the L1 and L2 cache are unified (i.e., contain both instructions and data). Such a configuration has been shown in [12] to perform as well or better on fork-join programs than dedicated instruction and data caches.

The L1 cache is 64 kB large, divided into four equally sized banks (16 kB each). The L2 cache is unbanked and 2 MB large. The L2 cache need not be banked as it is rare that there will ever be more than two L1 misses simultaneously, eliminating the need for banking as the L2 cache requires only one port.

2.1.2 Cache Power Model

Power is dissipated within a cache in two distinct ways. One way involves the dynamic energy spent to read or write a value to a cache location. In addition, static energy is spent to preserve the data already in the cache. For a cache level, *x*, the energy (in Joules) expended is:

$$E_x = DE_x + LE_x \tag{1}$$

where DE_x is the dynamic energy consumed by cache level *x* and LE_x the leakage energy consumed by *x*.

The overall dynamic energy of a cache is defined as:

$$DE = (N_{hit} + N_{miss} * 2) * de + (N_{hit} + N_{miss}) * ex \tag{2}$$

where *de* is the energy consumed for one memory access, *ex* the energy used in the crossbar in a multiported cache access (in a standard private cache, *ex* = 0), N_{hit} the number of cache hits (success attempts to access the cache), and N_{miss} is the number of cache misses (unsuccessful attempts to access the cache). Note that N_{miss} is multiplied by two in (2) because when a miss occurs, the memory is accessed twice, first when you check to see if the data is present in the cache resulting in a miss and also again to get the data value after it has fetched and placed in the cache.

The leakage power of a cache is:

$$LE = NA * lea \quad (3)$$

where *NA* is the number of cache blocks that are in use and not turned off while *lea* is the leakage energy of a cache block.

2.1.3 Processor Architecture

The processors in the core set are homogeneous (i.e. structurally identical) processing units with identical architectures and instruction sets. The processors consist of simple pipelines with six or more stages. Fig. 2.5 is one such example. This pipeline is similar to the one employed by MIPS processors [32]. The primary difference is that memory access are two cycles long. The crossbar stages used are described in section 2.2.1. Two memory access stages are required, the first to get the instruction from memory and the second to get any necessary data required by that instruction. The pipeline consists of six stages, the Instruction Crossbar (IX) where the crossbar is powered before the instruction is retrieved from memory, Instruction Fetch (IF) when the

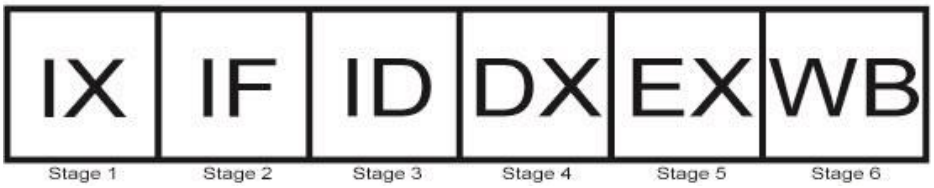

Figure 2.5 – Example six-stage pipeline for processors in ForPowER based on the standard MIPS pipeline. Stages one through six are Instruction Crossbar (IX), Instruction Fetch (IF), Instruction Decode (ID), Data Crossbar (DX), Execution (EX), and Writeback (WB), respectively.

instruction to be executes is fetched from memory, Instruction Decode (ID) where the instruction is decoded and any necessary data retrieved from the registers, Load/Store Crossbar (DX) where the crossbar for the data memory is powered, Execution (EX) when the instruction is executed, and Writeback (WB) where the data is written into the memory or appropriate register.

Most current processors are CMOS based [18]. Therefore, most of the power is consumed when the CMOS gates switch between logic true and logic false. This switching power is proportional to the clock frequency and the square of the supply voltage. As such, down scaling the frequencies of the processor's clock or the supply voltage can dramatically reduce the power of this CMOS logic (this is known as dynamic voltage scaling) [13].

An additional benefit of dynamically changing the frequency and voltages of the processors is that they will dissipate significantly less power. Similarly, if an area of the chip gets too hot, the clock speed of the processors in the area of the hotspot can be throttled to allow for the additional built up heat to be dissipated and then return to normal operation [39].

This approach to decreasing power consumption can have three specific drawbacks. First, in previous processor designs, dynamically changing the clock frequency took large amounts of time in comparison to overall job execution. However, it has been shown in [14] that current modern processors can switch from its current clock frequency to a target speed in only one clock cycle. This allows for negligible overhead between switching clock frequencies and supply voltage levels. Moreover, reducing the clock frequency inherently reduces performance, as clock frequency is

inversely proportional to the circuit delay [29]. This was considered unrealistic for many application spaces. This does not apply in fork-join parallelism because this reduction in performance of non-critical jobs actually has no effect on the overall net performance of the system, as long as their execution time is less than that of the critical job. As noted in Chapter 1, only the critical job needs to be run at maximum frequency. The slack available in the non-critical jobs allow those jobs to be run at a slower clock frequency, saving significant energy, as high as 90% [15]. Third, dynamically scaling the processor voltage and frequency requires additional hardware. The hardware required to modify the clock frequency is simple [38]. Dynamically scaling the processor supply voltage necessitates a few distinct hardware components and is more complicated than the hardware for scaling the frequency [38].

To effectively achieve maximal power savings, the different speeds at which the processors can run must be properly selected. Current processor designs permit 6-11 different processors speeds. Moreover, standard practice involves spacing each of these speeds equally for better granularity [16]. As such, we used in ForPowER a modified version of the Intel Pentium M 1.6 GHz processor, Intel's low power mobile processor, since it provides good performance, while running at significantly less power (only 10 Watts at maximum frequency) compared to similar CPUs [16]. We selected to allow it to throttle to clock speeds of 400 MHz, 600 MHz, 800 MHz, 1000MHz, 1200 MHz, and 1400 MHz, as this is the mapping of Intel's 1.6 GHz Pentium M onto to a maximum clock frequency of 1.4 GHz [17] .

2.1.4 Processor Power Model

As mentioned in section 2.1.3, modern processors are composed primarily of CMOS logic. Mudge in [18] defines the power consumption in CMOS logic circuits (in Watts) as:

$$P_{CMOS} = P_{Leakage} + P_{Static} + P_{Dynamic} \tag{4}$$

This equation consists of three components. First, the leakage power is the base amount of power lost due to leakage current regardless of the CMOS' gates or processors state (executing or idle). It is defined as:

$$P_{Leakage} = VI_{leak} \tag{5}$$

where V is the supply voltage and I_{leak} is the leakage current in the gates. Leakage power consists of approximately 2% of total power consumed in current generation CMOS logic [19].

The second component in CMOS power dissipation is static power, which as a result of the short circuit current I_{short} that flows between the supply voltage, V, and ground for a small time period, τ, when the CMOS gate's logic switches. Static power consumption is determined by:

$$P_{Static} = \tau AVI_{short} f \tag{6}$$

given that *A* is the average number of gates that switch per cycle. Static power currently accounts for 8% of total power consumption in processors [19].

The final term is the dynamic power consumption; it accounts for a majority of the power dissipation in CMOS, approximately 90% [19]. It is caused by the actual charging and discharging of the capacitive load, *C*, during each gate's operation. It is defined as:

$$P_{Dynamic} = ACV^2 f \tag{7}$$

For a given supply voltage, processors are run at a maximum clock, $f_{\max}$, so as to achieve the highest performance. f_{max} is related to the supply voltage, V, by the relationship:

$$f_{\max} \propto \frac{(V - V_{Threshold})^2}{V} \tag{8}$$

where $V_{threshold}$ is the threshold voltage of the CMOS gates that defines if it is in the high or low logic state. As such the maximum clock frequency is approximately linearly proportional to the supply voltage. It is then possible to simplify the dynamic power consumption to:

$$P_{Dynamic} \approx \alpha * f_{\max}^3 \tag{9}$$

where α is the combined coefficient from *A* and *C*.

When dynamic frequency scaling is performed on a processor, for the most part, it is only the dynamic power consumption that is affected. Therefore, only 90% of the overall power of the processor is scaled. As such, given the relationship in (9), you can determine the power utilized for a given f which is less than or equal to f_{max}. Therefore, the dynamic power can be rewritten:

$$.9 * P_{max} = \alpha * f_{max}^3 \tag{10}$$

If the processor undergoes dynamic frequency scaling and is run at a new frequency f ', which is less than or equal to f_{max}, the new value of the dynamic power, P', is:

$$P' = \alpha * (f')^3 \tag{11}$$

where P' is less than or equal to $.9P_{max}$. By dividing (11) by (10), we get:

$$\frac{P'}{.9 * P_{max}} = \frac{(f')^3}{f_{max}^3} \tag{12}$$

This then simplifies to:

$$P' = .9 * P_{max} * \frac{(f')^3}{f_{max}^3} \tag{13}$$

This means the power for a different clock frequency can be approximated knowing only the maximum clock frequency and the power it dissipates at that frequency. This significantly simplifies determining the power over any chosen frequency spectrum.

It is also important to note that when the processor power for a given speed has been determined, there are two distinct power profiles for the chip, one for when it is executing and the other for when it is idle (i.e. executing NOOP's, which are assembly language instructions that do not execute anything). The executing power is equivalent to the power determined by the maximum power and (13). The idle power consumption is simply 25% less than the power consumed at a given processor speed, while the processor is executing [20].

Moreover, it is necessary to note the fact that the percentages stated for each type of processor power applies to current generation processors. In future processors where the size of the logic gets increasingly small (e.g. 90 nm), the leakage power becomes more critical and can represent up to 50% of total processor power [36]. This would decrease the benefits due to dynamic frequency scaling.

2.2 Network Model

Within any SoC design, an interconnection fabric must exist that links all nodes together so they can communicate effectively. Within current designs, two design paradigms are in use, specifically shared medium architectures and point-to-point networks [33].

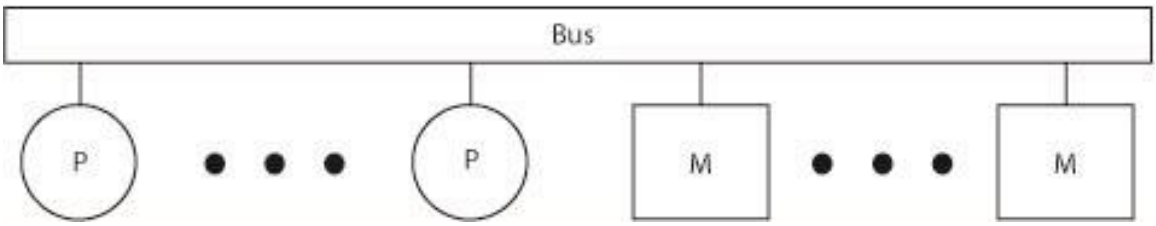

Figure 2.6 - A shared-medium network, where a bus connects multiple processors (P) and multiple memories (M).

First, shared medium architectures are the simplest of all interconnection models. In this model, all nodes share a single communication medium. Fig. 2.6 is an example of a shared medium network (in this case a bus) connecting multiple processors and memory devices. Since there is only one communication channel, only one node can drive the network at anyone time. This leads to a bottleneck as every other processor must stall until the network becomes free. Moreover, when a message is sent, it must reach all nodes along the communication channel before another message can be sent. As such, messages take significantly longer to travel through the network, since transmission continues even if the desired destination has already received the message. Shared medium networks are also extremely inefficient in low power architectures as every message must be broadcast to every single node, even if it is not the intended receiver, at greater energy cost [21].

Given the major drawbacks of a shared medium network, we decided to go with a point-to-point network. Point-to-point networks allow for connections between a single host and single receiver, reducing the cost to send each message as well as the time for it to propagate through the network. ForPowER's network consists of five routers. Each core set is given its own router that connects it to all the processors as well as the shared cache. Moreover, the scheduler is given its own dedicated router. In addition, each

router is connected to all of the other routers to allow for communication between all the core sets and the scheduler.

It has been shown in [22] for a network topology similar to the one we are using with similar load that the per byte cost to transmit data through the network is approximately 46.25 pJ.

Chapter 3 - Task Scheduler

This chapter details ForPowER's scheduler for efficiently assigning tasks to the different processing cores, previously presented in chapter 2. We will also outline the scheduling algorithm used by the scheduler and how it determines the processor speed for a job.

Since a large percentage of the overall power dissipation of an SoC can be dissipated by sending duplicate data over the network as well as powering this duplicate data multiple caches, an intelligent algorithm for assignment of the tasks to the individual cores is crucial. However, the assignment of the tasks to the independent processors must be done in such a way as to limit or negate any impact on the overall execution time of the program. The scheduler determines the best method for allocation of each task to its respective core. Tasks are assigned in their entirety to a single core and executed by that core till its completion (i.e., without preemption [34]).

An assumption, which is the predominant one in the literature [3], is also made here that the fork-join structure of the program is deterministic (i.e., the number of forks/parallel jobs is known *a priori*). Moreover, the number of parallel jobs never exceeds the number of processing nodes (although this model only describes 16 nodes meaning a maximum of 16 parallel tasks, this model has scalability to a larger number by adding more cores to a single cache or adding more core sets, if required by the program's structure). We assume that the expected execution time is known in advance to an acceptable level of accuracy.

The scheduler proposed in this thesis is based off of the fault-tolerant scheduler proposed in [5]. The scheduler determines all the scheduling variables for the entire

system including where to assign each task, the individual processors' clock frequencies and voltages, and where to synchronize all the jobs in a fork.

Fig. 3.1 shows the system's task scheduler. It has a dedicated processing unit to make decisions regarding where to route specific messages as well as determine the average execution times for each of the tasks. The execution time statistics for each of the tasks is stored within the scheduler. It is updated dynamically based on each task's execution time. There is a dedicated private cache for use by the scheduler's processor.

When the scheduler decides what data or instructions should be sent to a specific processor, a message is sent to the scheduler's block transfer engine to efficiently transmit blocks of data from either the instruction memory or the attached main memory that stores the source data. The block transfer engine (BTE) also breaks up large blocks of data into individual packets that can then be sent across the network. By having a dedicated BTE, it frees up the scheduler's processor to handle other necessary

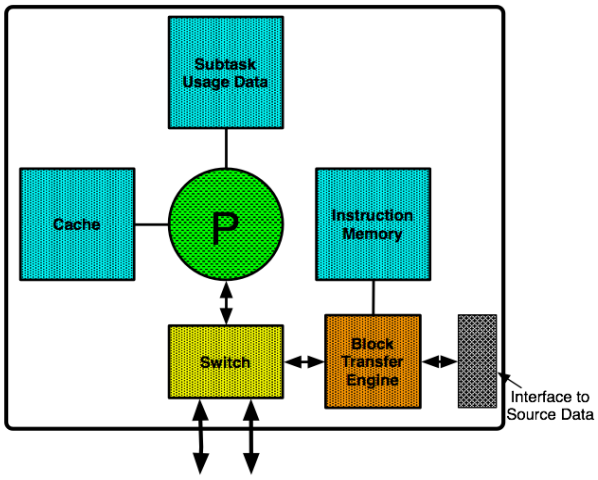

Figure 3.1 - The architecture's central scheduler consists of a single processor (P) with its own dedicated cache, a memory for storing task usage information for calculating expected execution time as well as a block transfer engine.

calculations, including updating the usage information of tasks as well as assigning them to cores.

The router in the figure is the interface of the BTE and processing unit to the network.

3.1 Task Assignment

Jobs are assigned to specific cores in the form of a queue. In a certain time period, a group of tasks, either from a fork and/or multiple parallel tasks, needs to be assigned. The scheduler determines the overall lowest cost associated with assigning a specific task to a specific processor; once that has been found, the data are sent by the block transfer engine to the cache associated with the processor, even before it has finished completing its previous execution. By sending it in advance, the time between the completion of the previous task and the beginning execution of the next is decreased to approximately only the time required to send data, if any, that is on another cache where a processor is still executing. The scheduler also sends the speed at which that processor should run for that task.

When assigning tasks to the nodes in this architecture, the cost is determined by four specific and distinct terms, with the overall cost for all tasks minimized. The assignment cost for task, *i*, on processor, *j*, is defined as:

$$c_{i,j} = (LocationReward)(ProcessorInUse + ForkCost + NeedtoTransferPenalty) \quad (14)$$

First, the location reward is the added benefit from staying within the same core set/processor instead of moving to a different location. By remaining on a processor attached to the same cache, the need to transfer data to another cache, which would waste time and energy in the network, is eliminated. Likewise, by staying in the same processor, assuming the next task is not a fork, there is no need to rescale the frequency or voltage of the processor itself. The reward is defined as:

$$LocationReward = 1 - s_c * \alpha - s_p * \beta \tag{15}$$

where s_c and s_p are binary terms representing whether processor j is in the same cache set and same processor as task the task whose execution preceded i, respectively. α and β are the rewards themselves for remaining in the same core set and processor respectively. These rewards are always greater than zero, and their sum is less than or equal to one.

ProcessorInUse is a penalty term from (14) that is used to prevent assignment of jobs to cores that already are in use and defined is as:

$$ProcessorInUse = M * s_j \tag{16}$$

where s_j is a binary term that represents whether processor j is currently idle or will finish executing before a certain preset time. M represents the Big M method in assignment issuing, where M is a disproportionately large positive number that is an overwhelming penalty to prevent any jobs being assigned to that processor [8].

ForkCost represents the expense of assigning task i to a core if it is the first task in a new fork. By keeping all of the jobs of a fork on as few core sets as possible, energy

that would have been dissipated by originally sending the instructions and data through the network to all the core sets, as well as sending the data back to a single core set to synchronize at the join, is significantly reduced. Moreover, to give later assignments more flexibility, the jobs with the higher expected execution times should be placed on the core set with the largest number of open nodes so the smaller jobs can be moved more easily, if necessary. As such, the cost for a fork is:

$$ForkCost = \frac{1}{\left(n_{o_j} * T_{\exp_i} * u\left(n_f\right)+1\right)} \tag{17}$$

where n_{oj} is the number of processor cores open on processor j's core set. T_{expi} is the expected execution time for the task i while n_f is the number of jobs in the new fork that is to be issued. u(n_f) represents the Heaviside unit step function:

$$u\left(n_f\right) = \begin{cases} 1, & n_f > 0 \\ 0, & n_f \le 0 \end{cases} \tag{18}$$

If the next task is not a fork, then the fork cost simplifies to one, and would be a standard cost across all cores.

The final term in (14) is the *NeedtoTransferPenalty* represents the need to move a specific job that is being executed on a core set that has a large number of open cores to one with less open core sets; the motivation for this is that it is better to leave core sets with as many open processors as possible to absorb a possible fork. It is inefficient to use only a small percentage of processors on a core set when there are nodes free elsewhere.

As such, it is beneficial to move these tasks to another core set. ForPowER uses a cost function defined as:

$$NeedtoTransferPenalty = M * u\left[\left(1 - n_f\right)\left(n_{o_j} - 3\right)\right] \quad (19)$$

This term is used to move a task off a core set only if it is not the first task in a fork, and the rest of the processor cores on *j*'s core set are idle. This term is easily adaptable to core sets with a different number of processors.

Given the cost function to assign a job *i* to processor *j* from (14), the cost to assign each job to each of the 16 processors is calculated. These costs are placed into a matrix with *R* rows (one row for each job in the fork) and 16 columns (one column for each processor). The cost, $c_{i,j}$, to assign job *i* to processor *j* is placed in the cell that corresponds to row *i* and column *j*. Using the Hungarian method on that matrix of costs, the optimal assignment of the *R* jobs to the 16 processors is determined [35].

3.2 Task Synchronization

Synchronization occurs when all jobs have completed execution. For synchronization to proceed, the required data from all the jobs needs to be sent to the processor assigned to complete synchronization. Our algorithm allows for all jobs to complete executing nearly simultaneously. As such, there is no time advantage to synchronization being assigned to any specific processor or core set. Therefore, the metric for synchronization should be to limit the amount of data that must be sent

between core sets to permit synchronization. We propose to define which core set is to synchronize based on the equation:

$$SynchronizationCost = 4 - n_a \tag{20}$$

where n_a is the number of jobs from the fork assigned to that core set. The scheduler can select any free processor on the core set with the lowest cost to perform the synchronization.

If a job is on a different core set than the one selected to synchronize, the data, upon completion of execution, is sent to the synchronization core set to await the join. If the job is on the same core set, then the processor simply becomes free again to be assigned another task to execute.

3.3 Processor Speed Determination

The final role of ForPowER's scheduler is to determine the processor clock frequency for the processors. As explained in Section 2.1.3, there are six different speeds for a processor to run at: 400 MHz, 600 MHz, 800 MHz, 1000 MHz, 1200 MHz, and 1400 MHz. The lower speeds consume less power than the faster speeds. Therefore, when a processor is idle and not in use, the scheduler will set its clock speed to as low as possible (400 MHz) as this will allow it to consume the least amount of energy.

However, when a processor is in use, it is necessary to be able to determine the proper processor frequency to save power without affecting the execution time of the overall program. Therefore, we need to assign the critical job to always run at the

maximum clock frequency as it inevitably will be the section of code that determines the overall execution time of a given fork.

Now knowing that the critical job will be running at the maximum clock frequency, we can determine the processor speed of all non-critical jobs. It was shown in [29] that for a job *k*, there is a required workload, w_k, which is the number of clock cycles needed to complete it. Similarly, the critical job has the maximum workload, w_{max}. Relative to the workload of the critical job, job *k*'s requisite clock speed, f_k (in MHz), can be determined by:

$$f_k = \left(1 + \left\lceil 6 * \frac{w_k}{w_{\max}} \right\rceil \right) * 200\,\mathrm{MHz} \tag{21}$$

Eq. (21) is derived from the linear relationship between execution time and workload for on-chip execution as explained in [29]. The workload ratio is multiplied by six because there are six possible processor speeds. The ceiling function is used in (21) since it is necessary to have the execution time of non-critical jobs be less than that of the critical job; therefore, the ceiling function gives non-critical jobs the lowest possible clock speed that still allows it to finish before the critical job. The whole sum is multiplied by 200 because that is the step-size between the different processor speeds.

Chapter 4 - Experimental Results

In this chapter, we describe the software we utilized to simulate both ForPowER and Kumar's architecture. Moreover, we present the benchmark selected to determine power usage for both architectures and compare the results. Finally, we discuss the sensitivities of ForPowER to different usage profiles.

When simulating a heterogeneous System on a Chip, there are two potential modeling paradigms that can be utilized. The first option to simulate is at the instruction level, which involves simulating the processing elements implementing every single instruction within a given application. However, these simulators will be inadequate for capturing the design tradeoffs in heterogeneous SoCs. This arises from the prohibitively large simulation times for instruction set simulation (ISS) as well as the time required to develop the ISS-level models.

To overcome the costs involved with simulating at the instruction level, it is possible to model heterogeneous SoCs at a higher level of abstraction. These higher layer simulators allow designers to manipulate threads, tasks, processors, and scheduling and communication strategies as opposed to instructions, functional units, and registers. These simulators have been shown to generate comparable results to those done at the instruction level in [23].

4.1 Software Implementation

We used the MESH (Modeling Environment for Software and Hardware) simulation suite to model both Kumar's architecture and ForPowER [24]. MESH is a

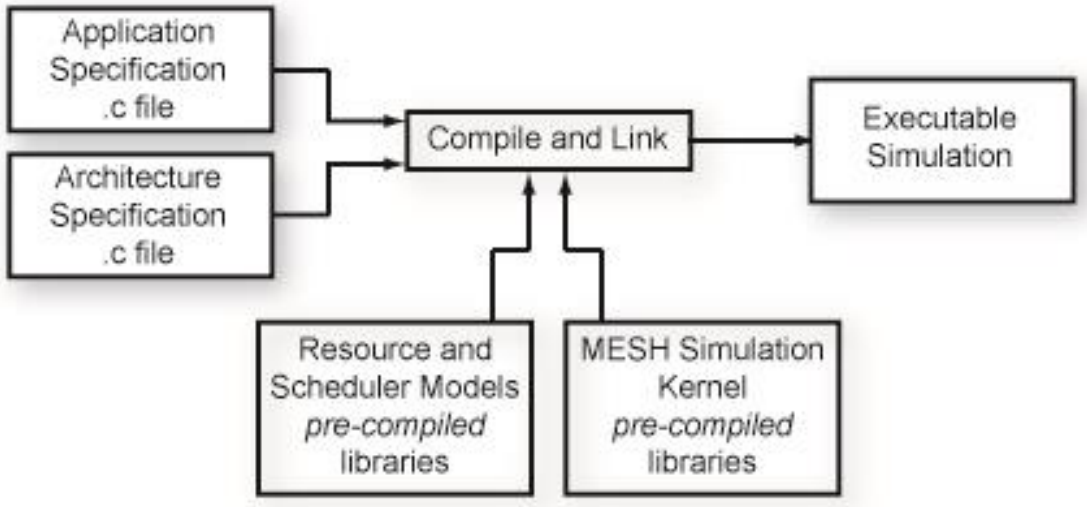

Figure 4.1 - MESH Simulator tool flow. The application and architecture specification files are compiled with the MESH libraries to get an executable simulation.

compiled simulator written in the standard C programming language. This allows for greater readability of simulation as the conventions and syntax of C are widely used and standard. Moreover, since most modern UNIX based machines have C compilers built into the operating system, the code is portable to UNIX-based machines.

Fig. 4.1 is the tool flow chart for the MESH suite. First, the designer specifies the program to be simulated as well as the architecture it will be run on within .C files. This is done through a set of application program interface (API) functions. These API's form the building blocks of the programs (e.g. forks, joins, threads, tasks) and the architectures (e.g. processors, schedulers, buses, etc.). The API's are provided by the precompiled resource and scheduler libraries that are part of the MESH suite. When the application and architecture .c files are compiled, they are linked with the MESH simulation kernels and the precompiled libraries forming a single executable file. [24]

4.2 Hydro2D Benchmark

There are numerous fork-join benchmarks that could be used to compare the energy consumed in the hardware of ForPowER to the energy consumed by Kumar's architecture. These include the fork-join benchmark that is part of parallel Java Grande Suite [37] and the NAS APPBT benchmark [7]. The benchmark we selected to use to test ForPowER was the Hydro2D benchmark. It was included as part of the Standard Performance Evaluation Corporation 1995 (SPEC95) benchmark package [7]. It solves the hydronamical Navier-Stokes equations to compute galactic jets. The benchmark is 390 kilobytes (kB) large [25] and requires a data set that is 8.71 megabytes (MB) [26].

Fig. 4.2 is the flow control graph of the Hydro2D benchmark. Each numbered or

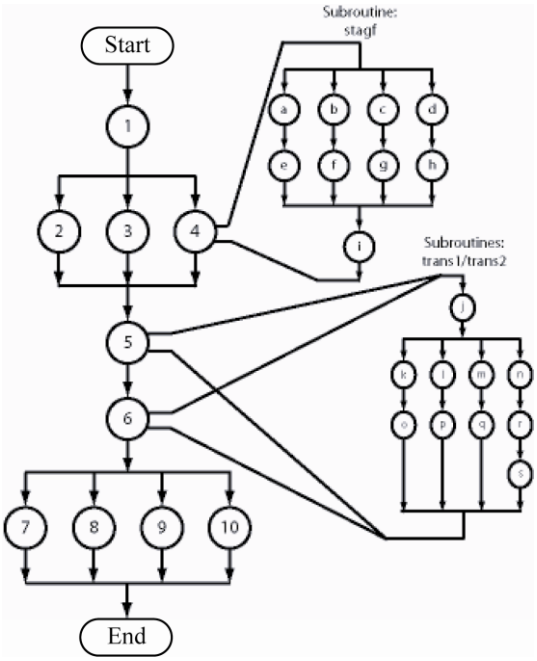

Figure 4.2 - Hydro2D Benchmark flow of execution where each circle represents a task with the number in the circle representing its task number. Tasks 4, 5, and 6 spawn another level of parallelism implementing the functions *stagf*, *trans1*, and *trans2*, respectively.

Table 4.1 – Hydro2D average cache miss rates per 1000 memory references

	Instructions Misses	Data Misses
16kB Cache	0.01	70
64 kB Cache	0.001	70

lettered circle represents a task that needs to be scheduled to a processor. Each of the tasks requires approximately the same execution time. Moreover, Hydro2D contains two levels of parallelism. After task 1 completes, it forks to three separate tasks each of which can be scheduled on their own processor. Task 4 then spawns (i.e., forks resulting in) the second level of parallelism where the subroutine, *stagf*, is run. At the completion of all the tasks, a join occurs, which must synchronize all jobs in that fork before execution can continue. Similarly, tasks 5 and 6 spawn additional parallelism for subroutines *trans1* and *trans2* respectively [7]. Both *trans1* and *trans2* have a single level of parallelism.

We assumed 90% instruction sharing and 11% data sharing between parallel jobs, as established in programs of this type in [11].

Hydro2D has 524 million instruction memory references and 195 million data memory references. Table 4.1 summarizes the cache miss rates for instruction and data references for Hydro2D. The two cache sizes (16kB and 64kB) are the cache sizes for Kumar's architecture ForPowER architecture respectively [12].

4.3 Experimental Results

We simulated the benchmark described in section 4.2 on both ForPowER and Kumar's architecture. We used a 16kB L1 cache and 512kB L2 cache for each processor for Kumar architecture and a 64kB shared L1 cache and 2MB shared L2 cache for ForPowER (resulting in the same amount of total memory); we used 70 nm technology for the transistor sizes in the caches. Since the same amount of total memory exists, the only additional hardware required for the caches in ForPowER are the four crossbars.

The dynamic power consumed for a single memory access [27] as well as the static power consumed per 32 byte block cache [11] in the respective caches is shown in table 4.2.

The power consumption in three components, namely, the processors, the caches, and the network, will be compared for both architectures. Table 4.3 summarizes the power dissipation in for both ForPowER and Kumar's architecture. All values of energy consumption are in millijoules.

The largest percentage and absolute savings in power occurred in the processors with over 70% decrease in power consumed by implementing dynamic voltage and

Table 4.2 – Kumar's and ForPowER cache dynamic and static power profiles

Kumar	Dynamic Power	Static Power
16 kB L1 Cache	113 pJ	0.4 pJ
512 kB L2 Cache	501 pJ	0.4 pJ

ForPowER	Dynamic Power	Static Power
64 kB L1 Cache	121 pJ	0.4 pJ
2 MB L2 Cache	950 pJ	0.4 pJ

Table 4.3 – Energy consumption (in mJ) of the Hydro2D benchmark on Kumar's architecture and ForPowER

	Kumar	ForPowER
Processors	264207	72511
L1 Dynamic	82.9	93.8
L2 Dynamic	6.8	13.0
Cache Static	34820	31256
Network	0.3	0.2
Total	299117	103874

frequency scaling.

In terms of cache power, ForPowER consumes slightly more dynamic power because of the need to power the crossbar in the L1 stage as well as the additional power per access for the L2 cache due to ForPowER's L2 cache's larger size. However, this small difference is made up by the substantial static power savings from the instruction and data sharing among the different jobs.

The power dissipated in the network is very small (less than .01% of all power consumed). ForPowER was still successful at saving power over Kumar in the network.

4.4 Sensitivity Analysis

When the amount of parallelism and sharing of instructions and data across the tasks is very high, the level of saving is greatest. In this section, we compare how ForPowER faired against Kumar's architecture under different workload conditions.

First, we utilized the same flow control graph and the size of data and instructions sets needed by the individual tasks from the Hydro2D benchmark (see Fig. 4.2) to test how ForPowER compared to Kumar's under different levels of sharing of instructions and data. To achieve this, we varied the level the sharing of instructions and data across the jobs in a fork. For simplicity, we assumed that the level of sharing of instructions and data was identical across all the jobs. This was simulated using the MESH suite described in section 4.2.

Fig. 4.3 shows the power consumed by the caches with different levels of sharing of instructions and data. With very low levels of sharing (less than approximately 1%), Kumar's architecture dissipated less energy than ForPowER since each L1 access in ForPowER has to also power the crossbar, while those in Kumar's do not, and each access to the L2 cache in ForPowER consumes more energy as shown in table 4.2. In section 4.3, the savings in static power from the elimination of the need to power redundant data made up for the difference in dynamic power. However, low levels of sharing do not provide sufficient opportunity for ForPowER to overcome this additional dynamic power, resulting in Kumar's architecture fairing slightly better. It should also be noted for even 100% sharing, there is still substantial power consumed by the cache as still one copy of each byte of data and instruction must be powered. This is the minimal amount of power that can be consumed for any benchmark with the same flow control and data as the Hydro2D.

In addition, there is a linear relationship between the savings in energy consumed by the cache versus different levels of sharing. This arises from the near uniform sharing of instructions and data in the tasks in the Hydro2D benchmark.

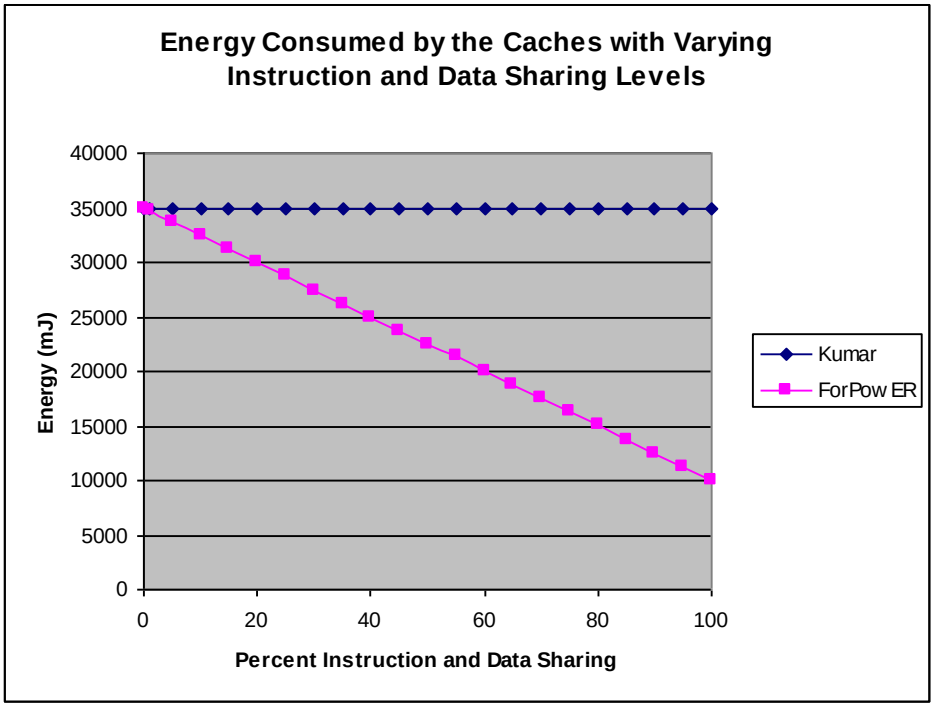

Figure 4.3 - Energy consumed (in mJ) by the caches in both architectures under varying levels of sharing of instructions and data

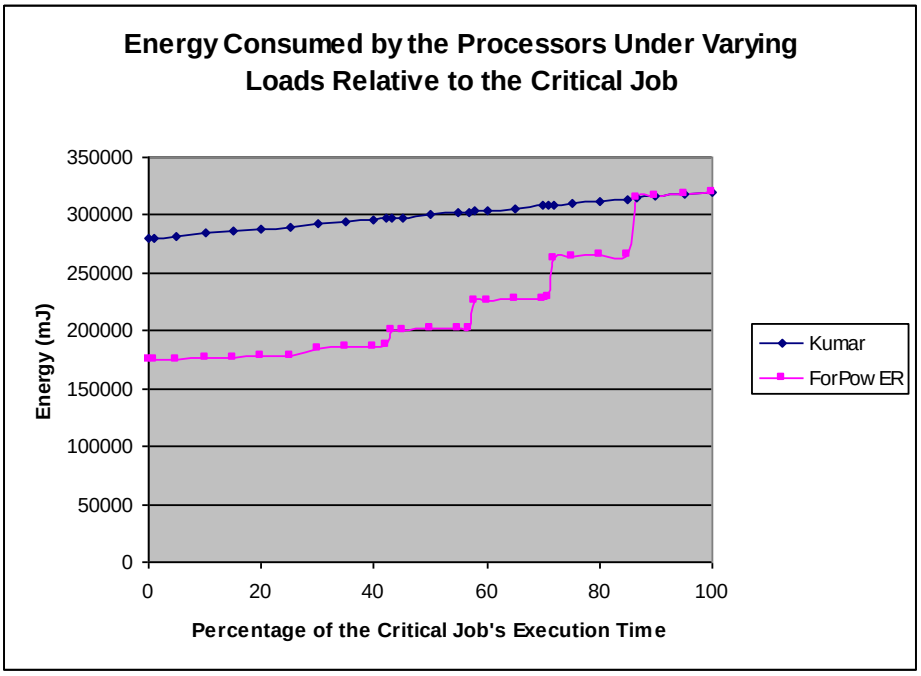

Figure 4.4 - Energy consumed (in mJ) by the processors in both architectures with varying processing loads relative to the critical job

Next, we tested ForPowER to see how it saved power relative to Kumar's architecture under various processing loads. To do this, we assigned eight of the processors identical jobs that would form the critical job; these jobs each required two seconds of execution time. This was kept static and used across the entire simulation. With the remaining eight cores, all were assigned equivalent jobs, whose execution time would be some percentage of the execution time of the critical job. The percentage of the critical job was varied to see the total energy consumed. This was all simulated using the MESH suite and results are in Fig. 4.4.

When the execution times of jobs were 85% or more of the critical job, ForPowER consumed the same amount of energy as Kumar's architecture. Over all other execution times relative to the critical job, ForPowER performed better, with a maximum savings of 38%. At no possible values did ForPowER perform worse than Kumar's.

Furthermore, at approximately 85% of the critical job, the biggest change in energy consumed occurred. This results from the change in processing speed from 1400 MHz to 1200 MHz for those executing non-critical jobs. A similar spike in the energy consumed occurs each time the job that is not the critical job can be completed in adequate time at the next lowest speed (for a list of the available processor speeds, see section 2.1.3).

Chapter 5 - Conclusions

This thesis proposed a power efficient architecture to exploit fork join parallelism using system on a chip, using total energy consumption as the metric. In Chapter two, we discussed ForPowER's design, describing its energy-efficient processors, caches and network. Chapter three described ForPowER's intelligent scheduling and synchronization algorithm.

Chapter four compared ForPowER's results with the results from Kumar's architecture, as presented in [3] to determine whose architecture consumed less energy. It was shown that in all categories tested (processor, cache, and network power) ForPowER outperformed Kumar's on the Hydro2D benchmark, leading to a total energy savings of over 65%. Moreover, we explained that for all but the lowest levels of data and instruction sharing (less than 1%), ForPowER's caches consumed less energy than Kumar's architecture. Furthermore, regardless of how long the execution time of each job is compared to the critical job, ForPowER always consumes less or the same amount of energy as Kumar.

The hardware required for the energy savings is only slightly more than that utilized in Kumar's architecture. The added hardware included the four crossbars for the shared caches as well as the hardware required for scaling the processors' clock frequencies and supply voltages. Moreover, our architecture was also able to utilize one less processor than Kumar's architecture by eliminating the need for a dedicated synchronization processor.

With the successful proof of concept, the next logical step is to synthesize this architecture to a series of field programmable gate array (FPGA) boards. This hardware

implementation scheme would allow for energy consumption results on actual hardware, while not necessitating special hardware construction.

Bibliography

[1] D. Culler and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, 3rd ed., San Francisco, CA: Morgan Kauffman, 2003.

[2] A. Jerraya and W. Wolf, *Multiprocessor Systems-on-Chips*, San Francisco, CA: Morgan Kauffman, 2005

[3] A. Kumar and R. Shorey, "Performance Analysis and Scheduling of Stochastic Fork-Join Jobs in a Multicomputer System," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 10, pp. 1147-1164, 1993.

[4] J. Lui, R. Muntz, and D. Towsley , "Computing Performance Bounds of Fork-Join Parallel Programs Under a Multiprocessing Environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 3, pp. 295-311, March 1998.

[5] M. Gergeleit, E. Nett, and J. Fitzner, "Online Prediction of Execution Times – A Basis for Adaptive Scheduling," *Proceedings of the Fourth International Workshop on Object-Oriented Real-Time Dependable Systems*, pp. 186-194, Jan. 1999.

[6] M.J. Quinn, *Programming in C with MPI and OpenMP*, 1st ed., New York: McGraw Hill, 2004.

[7] X. Martorell *et. al*., "Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors," *Proceedings of the 13th International Conference on Super Computing*, pp. 294-301, 1999.

[8] F. Hillier and G. Lieberman, *Introduction to Operations Research*, 8th ed., New York: McGraw Hill, 2005.

[9] T.Y. Morad, U.C. Weiser, and A. Kolodny, "Why Not Data Trace Cache," *Workshop on Duplicating, Deconstructing, and Debunking in Conjunction with the 32nd International Symposium on Computer Architecture*, June 2005.

[10] M.F. Mudawar, "Scalable Cache Memory Design for Large-Scale SMT Architecture," *Proceedings of the 3rd Workshop on Memory Performance Issues in Conjunction with the 31st Symposium on Computer Architecture*, pp. 65-71, 2004.

[11] L. Lin *et. al.*, "CCC: Crossbar Connected Caches for Reducing Energy Consumption of On-Chip Multiprocessors," *Proceedings of the Euromicro Symposium on Digital System Design*, pp. 41-48, 2003.

[12] M.J. Charney and T.R. Puzak, "Prefetching and Memory System Behavior of the SPEC95 Benchmark Suite," *IBM Journal of Research and Development*, vol. 41, no. 3, pp. 265-286, May 1997.

[13] YH Lu, L. Benini, and G. De Micheli, "Dynamic Frequency Scaling with Buffer Insertion for Mixed Workloads," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 11, pp. 1284-1305, 2002.

[14] F.R. Boyer *et. al.* "A Variable Period Clock Synthesis (VPCS) Architecture for Next-Generation Power-Aware SoC Applications." In *IEEE Proceedings of the 2nd IEEE Northeast Workshop on Circuits and Systems*, pages 145-148, 2004.

[15] L. Yuan and G. Qu, "Analysis of Energy Reduction of Dynamic Voltage Scaling-Enabled Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 12, pp. 1827-1837, 2005.

[16] "Intel PXA26x Processor Family Design Guide – Revision 1.0." October 2002. The Intel Corporation. <http://www.intel.com/design/pca/applicationsprocessors/ manuals/278639.htm>.

[17] "Intel Pentium M. Processor on 90nm Process with 2-MB L2 Cache Datasheet." January 2006. The Intel Corporation. <http://download.intel.com/design/mobile/ datashts/30218908.pdf>

[18] T. Mudge, "Power: A First-Class Architectural Design Constraint," *Computer*, vol. 34, no. 4, pp. 52-58, April 2001.

[19] J.M. Rabaey, A. Chandrakasa, and B. Nikolic, *Digital Integrated Circuits: A Design Perspective*, 2nd ed., Englewood Cliffs, NJ: Prentice Hall, 2002.

[20] V. Zyuban et. al., "Integrated Analysis of Power and Performance of Pipelined Microprocessors," *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 1004-1016, August 2004.

[21] L. Benini and G. De Micheli, "Networks on Chips: A New SoC Paradigm," *Computer*, vol. 35, no. 1, pp. 70-78, 2002.

[22] H. Wang, LS Peh, and S. Malik, "A Technology-aware and Energy Oriented Topology Exploration for On-Chip Networks," *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition*, vol. 2, pp. 1238-1243, 2005.

[23] B. Meyer et. al., "Power-Performance Simulation and Design Strategies for Single-Chip Heterogeneous Multiprocessors," *IEEE Transactions on Computers*, vol. 54, no. 6, pp. 684-697, June 2005.

[24] The MESH Group, *MESH User's Manual*, 0.04.09.27 ed., Pittsburgh, PA: Carnegie Mellon University, 2004.

[25] "Description of the Hydro2D Benchmark." *SPEC92 Floating Point Benchmark Package*. The Standard Performance Evaluation Company. 8 March 2006 <http://www.spec.org/cpu92/DESCR.090>.

[26] D. Burger, J. R. Goodman, and A. Kagi, "Memory Bandwidth Limitations of Future Microprocessors," *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 78-90, May 1996.

[27] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An Integrated Cache Timing, Power and Area Model", Technical report, Western Research Lab (WRL), Feb. 2001.

[28] F. Ho, A.S. Hou, and D.M. Bloom, " High-Speed Integrated Circuit Probing Using a Scanning Force Microscope Sampler," *IEE Electronic Letters*, vol. 30, no. 7, pp. 560-562, March 1994.

[29] K. Choi, R. Soma, and M. Pedram, "Dynamic Voltage and Frequency Scaling based on Workload Decomposition," *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pp. 174-179, 2004.

[30] E. Varki, "Response Time Analysis of Parallel Computer and Storage Systems," *IEEE Transactions of Parallel and Distributed Systems*, vol. 12, no. 11, pp. 1146-1161, Nov. 2001.

[31] H. Wang, LS Peh, and S. Malik, "Power-driven Design of Router Microarchitecture in On-Chip Networks," *Proceedings of the 36th International Symposium on Microarchitecture*, pp. 105-116, 2003.

[32] D. Patterson and J. Hennessey, *Computer Organization and Design - The Hardware/Software Interface*, 3rd ed., San Francisco, CA: Morgan Kaufmann, 2005.

[33] J. Kurose and K. Ross, *Computer Networking - A Top-Down Approach Featuring the Internet*, 3rd ed., Boston: Addison Wesley, 2005.

[34] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts*, 7th ed., Danvers, MA: John Wiley & Sons Inc., 2005.

[35] L.R. Foulds, *Combinatorial Optimization for Undergraduates*, New York: Springer-Verlag, 1984.

[36] S. Narendra et. al., "Full-Chip Subthreshold Leakage Power Prediction and Reduction Techniques for Sub-0.18-/spl mu/m CMOS," *IEEE Journal of Solid-State Circuits*, vol. 39, no. 3, pp. 501-510, March 2004.

[37] L.A. Smith, J.M. Bull, and J. Obdrizalek, "A Parallel Java Grande Benchmark Suite," *ACM/IEEE 2001 Conference on Supercomputing*, pp. 1-10, Nov. 2001.

[38] R. Ghattas and A.G. Dean, "Energy Management for Commodity Short-Bit-Width Microcontrollers," *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 32-42, 2005.

[39] W.R. Daasch, C.H. Lim, and G. Cai, "Design of VLSI CMOS Circuits Under Thermal Constraint," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 49, no. 8, pp. 589-593, Aug. 2002.

Appendix A – Nomenclature

α : Reward in the assignment algorithm for scheduling a job to continue executing on the same core set

β : Reward in the assignment algorithm for scheduling a job to continue executing on the same processor

A : Average number of CMOS gates that switch states per cycle

ALU : Arithmetic Logic Unit

API : Application Program Interface

ASIC : Application specific integrated circuit

BTE : Block transfer engine

C : CMOS gate's capacitive load

CPU : Central processing unit

de : Energy consumed by the cache in one memory access

DE_x : Total dynamic energy consumed by cache level *x*

DSP : Digital Signal Processor

ex : Energy consumed by the crossbar in one memory access

f : Processor clock frequency

f_{max} : Processors maximum clock frequency

ForPowER : The Fork-join Power Efficient aRchitecture

GHz : Gigahertz

I_{short} : Short circuit current

ISS : Instruction set simulation

I_{leak} : Leakage current of the CMOS logic

kB : Kilobyte

L1 : Level one of the cache

L2 : Level two of the cache

lea : Leakage energy of a cache block per cycle

LE_x : Total leakage energy consumed by cache level *x*

MB : Megabyte

MESH : Modeling Environment for Software and Hardware

MHz : Megahertz

n_{oj} : Number of processors open on processor *j*'s core set

N_{hit} : Number of cache hits

N_{miss} : Number of cache misses

NOOP : No operation

r_k : Number of tasks required to complete job *k*

R : Number of jobs in a fork

S : Synchronization processor in Kumar's architecture

SoC : System on a chip

T_{expi} : Expected execution time of task *i*

u(x) : Heaviside's unit step function, if x is greater than 0, then u(x) = 1, otherwise u(x) = 0

V : Supply voltage of the processor

$V_{threshold}$: Threshold voltage of the transistor which determines if it is in the high or low state