High-Level Power Analysis for Multi-Core Chips

Noel Eisley, Vassos Soteriou, and Li-Shiuan Peh
Dept. of Electrical Engineering, Princeton University
Princeton, NJ 08544
{eisley, soteriou, peh}@princeton.edu

ABSTRACT

Technology trends have led to the advent of multi-core chips in the form of both general-purpose chip multiprocessors (CMPs) and embedded multi-processor systems-on-a-chip (MPSoCs), with on-chip networks increasingly becoming the de facto communication fabric between cores as the demand for on-chip bandwidth scales up. These multi-core chips are composed of two key subcomponents: processor cores and a network fabric. Rapid, early-stage power estimation of these multi-core chips is crucial in assisting compilers in determining the most efficient thread partitioning and placement. While prior work in high-level power analysis exists, the focus has been on uniprocessor cores and ignores the interactions between cores via the on-chip network, as well as the power contribution of the on-chip fabric itself. In this paper we propose a first high-level power analysis framework that synergistically considers both computation and communication in a complete CMP system. Processor cores and the communication fabric are both abstracted as network nodes and links, so data dependencies, structural dependencies and communication dependencies are all modeled as resource contention, with resource utilization as a proxy for relative power. Our tool has been validated against the cycle-accurate BTL simulator of the MIT Raw CMP, showing an average speedup of 7X while achieving relative accuracy of 9.1%. We see this as a first step towards enabling the implementation of parallelizing compilers that explore various power-performance tradeoffs for future multi-core chips.

Categories and Subject Descriptors: B.8.2 [Performance and Reliability]: Performance Analysis and Design Aids; C.2.0 [Computer-Communication Networks]: General

General Terms: Design, Theory

Keywords: Power analysis, multi-core, chip multiprocessor (CMP), system-on-a-chip (SoC), simulation

1. INTRODUCTION

Power has become the most critical constraint in the design of many systems, from high-performance servers to embedded battery-operated devices. Recognizing the need to target increasing power consumption and design complexity in these systems, designers have turned to multi-core architectures such as chip multiprocessors (CMPs) [15, 26, 34] and multiprocessor systems-on-a-chip (MPSoCs) [1, 24]. As the demand for bandwidth rises in these on-chip systems, multi-core chips are increasingly turning to communication fabrics in the form of on-chip networks or networks-on-chips[1] (NoCs) that effectively scale to handle this increasing communication demand among the cores. Consequently, these multi-core chips are composed of two key subsystems: computation (processor cores) and communication (NoCs).

Rapid early-stage estimation of both subsystems' power contribution is critical as the communication subsystem not only consumes significant power [2, 40] but also substantially influences the activity on the various computation cores, with network messages functioning as synchronization points between cores. Quick estimates of the effect of different application partitionings on the power profile of a multi-core chip are therefore useful in guiding parallelizing compilers as well as for carrying out rapid design-space exploration. There is thus a need for fast, high-level power estimation tools for multi-core chips that synergistically consider processor cores and NoCs.

The challenges encountered in high-level power estimation are multi-fold. First, a temporal power profile is necessary; in other words, the power consumption trends of a multi-core chip across time needs to be captured. Estimating just average power consumption is insufficient as power consumption can vary significantly across time [12, 22] and constraining maximum power is of high interest as it impacts power delivery circuits [28]. The battery lifetime of portable systems can also vary by as much as 20% depending on the power consumption profile as a function of time [22]. For high-performance multi-core chips, the chip temperature depends critically on the average power dissipation across the thermal time constant. Second, speed is crucial for evaluating cost-benefit tradeoffs across a large array of designs. This is particularly challenging with multi-core chips, where the complexity of power estimation scales substantially beyond that of uniprocessors. Third, the power estimation needs to

[1]NoCs tend to refer to heterogeneous fabrics for MPSoCs while on-chip networks are usually used to refer to homogeneous fabrics for CMPs. In this paper we do not make this distinction, and we use these terms interchangeably.

maintain relative accuracy so that the power efficiency of alternative design choices can be accurately traded off. Finally, the tool needs to be readily extensible to cover myriad future designs, an especially important feature in multi-core chips where diverse architectures have been proposed.

In this work, we demonstrate a high-level power analysis framework for multi-core chips that is based on LUNA [10] (Link Utilization for Network power Analysis), a high-level power analysis framework for NoCs that has since been demonstrated to be effective for power-aware compiler optimizations in networks [33], and has been incorporated within the SUIF2 parallelizing compiler for network power estimations and optimizations [5, 6]. In the work by Soteriou et al. [33], LUNA is employed by the compiler to generate fast high-level estimates of the expected temporal and spatial power profiles in the network. These estimates are used to generate directives which are stored at each router and direct the operation of dynamic-voltage-scalable (DVS) links, facilitating rapid responses to changing network traffic conditions. To the best of our knowledge, ours is the first work to model both processor cores and the communication fabric at a high level within the same analysis framework. Here, both processor pipelines and networks are abstracted as nodes and links. More specifically, processor pipeline stages and network routers are modeled as nodes, and the data flows between processor structures and network interconnects are modeled as links. Our results show that our tool is (1) rapid: it is close to an order of magnitude faster when compared to the Raw BTL simulator [35], a cycle-accurate multiprocessor simulator running a range of SPEC benchmarks, with an average speed of 1.2 billion instructions per hour when simulating a 16-tile multi-core system[2]; (2) relatively accurate: it exhibits 9.1% relative accuracy on average when validated against the same simulator; and (3) extensible: it has been used to model a range of processor cores and networks, where both homogeneous CMPs and heterogeneous MPSoCs can be modeled.

The rest of this paper is organized as follows. In Section 2, we present a brief background on LUNA. Following, in Section 3, we propose how processors can be mapped onto nodes and links and then present in detail our high-level power analysis for processors. Section 4 combines the processor-network frameworks and proposes a new framework for high-level power analysis of complete multi-core chips. Our results are validated against a detailed model of the MIT Raw CMP that was verified against actual silicon. Section 5 discusses related work, and finally Section 6 concludes the paper.

2. HIGH-LEVEL NETWORK POWER ANALYSIS

Our proposed power analysis for multi-core chips rests upon modeling the entire CMP/MPSoC as a network. Both the processor cores as well as the NoC communication fabric are modeled as a fully connected directed graph $G = (N, L)$ where N is the set of nodes and L is the set of links in G. This enables us to then leverage an existing high-level network power analysis framework, LUNA [10].

LUNA was chosen as the foundation of our complete chip power analysis framework for a number of reasons:

- LUNA captures spatial and temporal variability in the power profile.
- LUNA is fast, with a runtime shown to be up to 360X faster than Orion [40], a cycle-accurate network power simulator, as it is an analysis rather than a simulation-based framework.
- LUNA preserves relative accuracy, which was shown to be within 5.9% of Orion.
- LUNA is extensible and has been shown to model a wide range of communication fabrics with ease, as it does not require the explicit modeling of each component's functionality as in most cycle-accurate simulators.
- LUNA has also been shown to be effective in enabling compiler-driven power optimizations of the network fabric [33], and it has been incorporated into the SUIF2 parallelizing compiler framework [6, 5].

2.1 Link Utilization as a Proxy for Power

Figure 1 shows the microarchitecture of a typical on-chip network router consisting of buffers, an arbiter, a crossbar, and a set of links. Dynamic network power, P_N, is a function of activity/utilization and energy costs (constants) for each of these key router components:

$$\begin{aligned} P_N \; = \; & \sum_{j=1}^{4N}(U_{link_j} \cdot E_{link} + U_{bufwr_j} \cdot E_{bufwr} + \\ & + U_{arb_{\lfloor j/4 \rfloor}} \cdot E_{arb} + U_{bufrd_j} \cdot E_{bufrd} + \\ & + U_{xbar_{\lfloor j/4 \rfloor}} \cdot E_{xbar}) \end{aligned} \qquad (1)$$

where N is the number of nodes in the network, j is the enumerated index of each link (four links per router in a typical mesh), and $E_{[component]}$ is the energy cost for a major component of the router [40].

LUNA uses link utilization as an abstraction for network power; in other words, the level of activity at a network link is used as a proxy for the overall power consumption of that network router and link, reducing Equation 1 to

$$P_N \; = \; \sum_{j=1}^{4N} U_{Link_j} \cdot (E_{link} + E_{bufwr} + E_{arb} + E_{bufrd} + E_{xbar}) \quad (2)$$

Such an abstraction works for high-level network power analysis because each flit[3] of a packet will definitely incur the various energy costs: it will traverse the incoming link, be written into the buffer, go through arbitration(s)[4], be read out from the buffer when granted crossbar traversal, and finall traverse the crossbar. Since NoCs typically do not drop packets, the activity at a link is a strong indicator of the internal activity of the router across all of its components (buffers, crossbar, etc.) as the two sets of activity, internally in the router and at the link are almost identical. Thus, the link utilization is a good proxy for measuring dynamic power. As described in the work by Eisley and Peh [10], leakage power can also be modeled by adding a fixed cost to link utilization, or by lowering the link capacity by a constant factor.

[2]This speedup estimation is based on 1,000-cycle sampling periods, applied to a CMP with 16 processing tiles abstracted as a 272 node, 464 link network-graph.

[3]Flit stands for flow control unit, a fixed-size segment of a packet.

[4]Only the number of arbitrations will differ depending on the level of network contention. However, it has been shown that arbiter power is a small fraction of total router and link power [40].

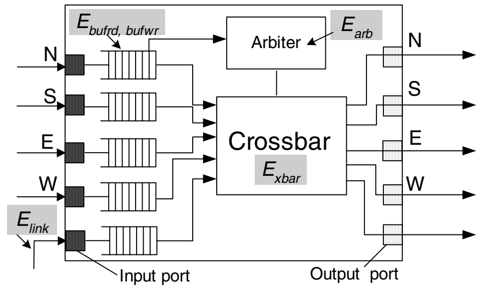


Figure 1: Microarchitecture of a typical NoC router and its associated energy costs for each component.

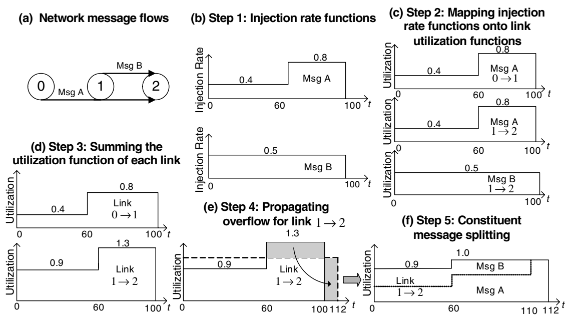


Figure 2: A walkthrough example showing LUNA's five-step link utilization analysis. Time t is in terms of clock cycles.

Clearly, abstracting (network) power as resource (link) utilization ignores many detailed components of power, such as the effect of bit-level switching activity on dynamic power, and that of input state on leakage power. However, this approach enables a stateless analysis framework that is much faster than cycle-accurate architectural power simulations [40] while maintaining good relative accuracy [10].

2.2 How LUNA Analyzes Link Utilization

With the aid of Figure 2 we explain in detail LUNA's five key steps of analysis for deriving the link utilization of every network link across time, when fed with a specific network traffic. For clarity, we use a small three-node network whose nodes are enumerated as 0, 1, and 2 (see Figure 2(a)).

Step 1. In the first step, message traffic between nodes is captured as *injection rate functions*, with the injection rate of each message expressed in units of flits per cycle. These injection rate functions can be obtained by integrating LUNA into the compiler flow: as the compiler maps the code to the processors of a multi-core chip, it uses knowledge of the instruction dependencies and explicit message-passing instructions along with the source and destination of each dependency to calculate the injection rate functions. Figure 2(b) shows the injection rate functions of two message flows A and B over the first 100 clock cycles.

Step 2. In this step, the injection rate functions of Step 1 are mapped onto links of a network topology as specified by the routing algorithm, translating them into *link utilization functions*, again measured in terms of flits per cycle. Figure 2(c) shows how message flows A and B are both routed along links $0 \rightarrow 1$ and $1 \rightarrow 2$ for the same time duration of 100 clock cycles.

Step 3. Next, for each link, all utilization functions which have been mapped to that link are *superimposed* and *summed*, to reflect the sharing of links amongst multiple message flows in a network. Figure 2(d) shows that this summation detects traffic contention or overflow in link $1 \rightarrow 2$ between cycles 60 to 100 since the normalized utilization rate of 1.0 is exceeded (here we assume that the maximum link bandwidth is one flit per cycle).

Step 4. To account for link contention, LUNA *propagates* this overflow as depicted by the shaded area in Figure 2(e). Intuitively, this overflow area corresponds to the number of flits that have to be buffered and transmitted later as they exceed the maximum link capacity.

Step 5. Finally, the link utilization functions are *split* back into *constituent* message flows to reflect how individual message flows are affected by the contention. Fair arbitration is assumed in splitting the link utilization among the message flows as shown in Figure 2(f).

These steps are repeated unidirectionally along the temporal axis until no more traffic is injected into the network and there is no remaining traffic that has been held back due to contention. The resulting link utilization functions provide an estimation of network power across space and time.

3. HIGH-LEVEL PROCESSOR POWER ANALYSIS

Here, we model a processor core's architecture as a network so as to enable us to utilize LUNA's framework to analyze processor power consumption. Modeling of the processor architecture is carried out as follows: each resource of the processor (functional units, cache, register file, etc.) is mapped to a link in the network so that the utilization of each resource becomes a link utilization in the network and thus forms a proxy for its power consumption. The nodes in our network merely exist to connect the links (resources) in an orderly manner to reflect the processor microarchitecture and its pipeline.

Our modeling consists of two phases: first the processor microarchitecture is mapped onto a network; and second the instructions are mapped (translated) to network messages. Formally, our framework performs the following mappings:

$$R \rightarrow L \times N \tag{3}$$

$$I \rightarrow M \tag{4}$$

where R is the set of resources, such as the functional units of the microprocessor to be modeled, L is the set of links in the network, N is the set of nodes in the network, I is the set of instructions in the assembly program, and M is the set of messages injected into the network. Note that the user decides upon the actual resources of the microprocessor to be modeled and the depth of detail of the microprocessor model. We briefly discuss the issue of the appropriate level of detail in Section 3.5.

3.1 Resource Utilization as a Proxy for Power

Similarly to the way we abstract network power through link utilization in Section 2, we abstract the power consumption of a processor by the utilizations of individual resources. The summation of the energy costs of each component (functional units, register file, caches etc.) is captured by each respective utilization function as follows:

$$P_{Proc} = \sum_{i=1}^{|component|} U_{component_i}(t) \cdot E_{component_i} \tag{5}$$

Fundamentally, Equation 1 is a specific case of Equation 5 where $E_{component} = \{E_{link}, E_{bufwr}, ...\}$. However, there is a significant difference between the way we use Equations 5 and 1. In the case of the network fabric, the utilization of all of the components is approximately equal because message flows in networks consume roughly the same amount of energy per hop. Estimates of individual network components' energy consumptions ($E_{link}, E_{bufwr}, ...$) are thus not necessary because P_N is a relative power measure; constant factors can be eliminated. However, in the case of a processor pipeline, each instruction will not consume the same amount of energy. The component utilizations hence cannot be removed and abstracted as a single utilization in Equation 5. Relative estimates of $E_{component_i}$ are thus required in order to obtain a relatively accurate estimation of P_{proc}.

Therefore the additional complexity exhibited here in the case of a processor modeled as a network as compared to explicit NoC modeling in Section 2 centers around the issue of how to obtain relatively accurate energy costs for each processor resource. One way is to use capacitance estimates, since $E \propto CV^2$. The most accurate capacitance estimates are derived from low-level tools which are typically run just prior to the tape-out of a processor. However, good estimates of capacitance for individual components can be obtained prior to fabrication, even at the early stages of design, based on previous experience and similar existing designs and technologies. Furthermore, since capacitance is in turn a function of area, relative energy can be estimated by the area of the die that each (processor or network) component occupies and it can be scaled across process technologies. In short, while exact estimates can only be obtained from late-stage designs, good estimates of $E_{[component]}$ can be obtained from early-stage designs and similar existing designs. Coupled with the $U_{[component]}$ values analyzed and derived by LUNA, we can then derive a relatively accurate estimate of the power consumed by each component of the processor core across time. Section 4.2 investigates and demonstrates that our framework is not overly sensitive to the accuracy of these relative energy estimates, affirming its feasibility as an early-stage power analysis tool.

3.2 Phase 1: Mapping a Processor Microarchitecture onto a Network

As an illustration, we show a block diagram of the microarchitecture of a single processor core within the MIT Raw CMP [34] that we model in Figure 3(a). Figure 3(b) depicts the corresponding network-mapped representation. Figure 3 is also used as the basis of our walkthrough example in Section 3.4. The Raw chip is described in more detail in Section 4. We have also modeled three other processor cores as networks in a similar way using LUNA: the Intel Pentium Pro, the Alpha 21264 and the MIPS R10000 cores, but details of their modeling are omitted here for brevity.

In Figure 3(b), we see how the overall structure of the mapped network emulates the pipeline flow of the processor. Starting from the left, there is the instruction fetch, followed by the instruction decode stage, followed by the register fetch stage. Continuing to the right of Figure 3(b), instructions flow either through the functional units or, in the case of memory reads or writes, through the address (`A`), tag lookup (`TL`), and tag verify (`TV`) stages, and finally through the writeback stage.

Our framework allows the user to design a custom model;

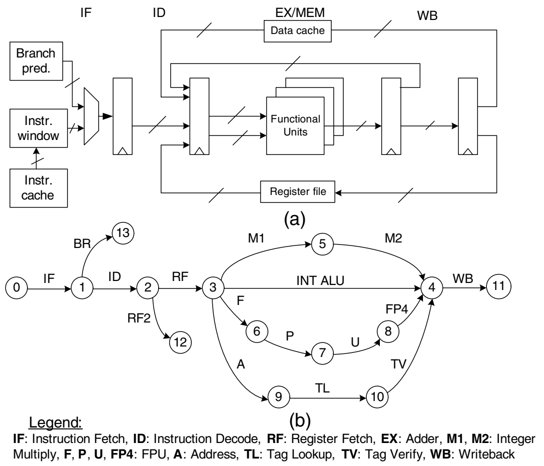

IF
ID
EX/MEM
WB
Data cache
Branch pred.
Instr. window
Instr. cache
Functional Units
Register file
(a)
BR
IF
ID
RF
RF2
M1
M2
INT ALU
F
P
U
FP4
A
TL
TV
WB
0
1
2
3
4
5
6
7
8
9
10
11
12
13
(b)
Legend:
IF: Instruction Fetch, ID: Instruction Decode, RF: Register Fetch, EX: Adder, M1, M2: Integer Multiply, F, P, U, FP4: FPU, A: Address, TL: Tag Lookup, TV: Tag Verify, WB: Writeback

Figure 3: (a) Block diagram of a typical microprocessor architecture and (b) its corresponding network representation. The numbers here represent node indices which are referred to in the walkthrough.

here we present a set of general rules, using the processor pipeline of Figure 3(a) and its corresponding network representation of Figure 3(b) as an illustrative example:

1. **Base network.** We begin with a single chain or path of nodes and links where each link represents a stage in the processor pipeline. Therefore, if the processor has an N-stage pipeline, there will be N links and $N + 1$ nodes. This base network is depicted in the form of a path $0 \to 1 \to 2 \to 3 \to 6 \to 7 \to 8 \to 4 \to 11$, in Figure 3(b); it is a 9-node path for the 8-stage Raw pipeline.
2. **Instruction fetch.** This represents an access to the instruction cache: $0 \to 1$.
3. **Instruction decode.** Since each instruction must go through the decode process, a single path sufficiently models this stage: $1 \to 2$. For conditional branches, an additional access to the branch predictor is needed, requiring another link: $1 \to 13$.
4. **Register fetch.** Here, either one or two register values are read. So there are two paths through the register fetch stage corresponding to register reads: $2 \to 12$ and $2 \to 3$.
5. **Instruction execute.** Through the EX/MEM stage, instructions either pass through one of the functional units ($3 \to 4$, the integer execution unit; $3 \to 5 \to 4$, the integer multiply unit; or via path $3 \to 6 \to 7 \to 8 \to 4$, the 4-stage Raw floating point unit) or make an access to the data cache via path $3 \to 9 \to 10 \to 4$. The general trend here is that each possible path through the processor pipeline is a separate path through our network. The length of each ALU path is determined by the latency of that ALU (the integer multiply is a 2-cycle operation and hence is mapped onto a 2-hop path $3 \to 5 \to 4$).
6. **Writeback.** Finally, the last stage is the writeback stage: $4 \to 11$. Note that in most cases, even if a result from an instruction is used by a successive instruction

addiu r3 r1 imm
messages 3 intALU
fet+0 src 0 dest 2 0 1 2 (IF,ID)
iss+0 src 2 dest 3 2 3 (R1 read)
iss+1 src 3 dest 11 3 4 11 (EX,WB)
addu r3 r1 r2
messages 4 intALU
fet+0 src 0 dest 2 0 1 2 (IF,ID)
iss+0 src 2 dest 3 2 3 (R1 read)
iss+0 src 2 dest 12 2 12 (R2 read)
iss+1 src 3 dest 11 3 4 11 (EX,WB)

j imm
messages 2 NA
fet+0 src 0 dest 2 0 1 2 (IF,ID)
iss+0 src 3 dest 4 3 4 (EX)
sw r1 r2 imm
messages 3 LDST
fet+0 src 0 dest 2 0 1 2 (IF,ID)
iss+0 src 2 dest 3 2 3 (R1 read)
iss+0 src 2 dest 12 2 12 (R2 read)
iss+1 src 3 dest 11 3 9 10 4 11 (EX,WB)

add.s r3 r1 r2
messages 4 fpALU
fet+0 src 0 dest 2 0 1 2 (IF,ID)
iss+0 src 2 dest 3 2 3 (R1 read)
iss+0 src 2 dest 12 2 12 (R2 read)
iss+1 src 3 dest 11 3 6 7 8 4 11 (EX,WB)
mulh r3 r1 r2
messages 4 intMUL
fet+0 src 0 dest 2 0 1 2 (IF,ID)
iss+0 src 2 dest 3 2 3 (R1 read)
iss+0 src 2 dest 12 2 12 (R2 read)
iss+1 src 3 dest 11 3 5 4 11 (EX,WB)

Figure 4: Partial Raw instruction set architecture (ISA) description. `addiu`: adds r1 and imm and stores the result into r3; `addu`: adds r1 and r2 and stores the result in r3; `j`: jumps to instruction at memory address imm; `sw`: stores the value in r1 at the memory address held in r2 plus the offset (imm); `add.s`: adds r1 and r2 and stores the result to r3; `mulh`: multiplies r1 and r2 and stores the upper 32 bits to r3. The various numbers, for example, "dest 2 0 1 2," refers to the nodes of the network depicted in Figure 3. The text contained in parentheses above aids the reader in mapping individual messages to processor pipeline stages.

via the data forwarding path, it still must continue through the writeback stage.

7. **Link bandwidth.** With the topology of the "network" mapped, the final step involves setting the bandwidth of each link in this mapped network. This is equal to the pipeline width for a corresponding component in terms of instructions per cycle. Thus, if the processor is capable of fetching four instructions per cycle, the bandwidth of the $0 \rightarrow 1$ link in Figure 3(b) is four. Similarly, if there are two identical integer multiply units, then the bandwidth of links $3 \rightarrow 5$ and $5 \rightarrow 4$ is two. Since the Raw processor is a single-issue machine, all of the links in Figure 3(b) have bandwidth of one.

3.3 Phase 2: Mapping Processor Instructions to Network Messages

Each processor instruction is mapped to one or more messages in our network. These message-mapped instructions are then injected into LUNA's framework. We make use of the example in Figure 4 to show a partial ISA description (in this case using Raw's ISA) where the instructions `addiu, addu, j, sw, add.s`, and `mulh` are given.

The inherent flexibility exhibited by our architecture model leads to the customization of the way instructions are mapped onto messages. While Section 3.4 explains Figure 4 in detail, here we first describe a set of general rules which we use in this work and which can be incorporated into the modeling of any architectural model.

1. All instructions have a message which starts at node 0 (instruction fetch) and ends at node 3 (after register fetch but before execution). As mentioned previously, this is due to the fact that all instructions must be fetched from instruction memory and decoded.
2. An additional message is required from node 3 to node 11. This represents the instruction passing through the EX/MEM stage and later through the writeback stage. The reason for splitting an instruction into two messages, $0 \rightarrow 3$ and $3 \rightarrow 11$, is that instructions may be fetched and decoded, but then may not be able to enter execution because they may be either waiting on previous instructions to finish execution and write back their results (data dependencies), or they may be waiting for functional units to become free (structural dependencies). Incorporating a single message from node 0 to node 11 (from fetch to commit) prevents this.
3. Additional resources require further messages which branch out from the two main messages per instruction as described above. For example, if the instruction is a branch it will then traverse the branch predictor path. To model the branch prediction, a message is injected from node 1 to node 13.

Dependency modeling and temporal analysis. The injection time of each message (equivalently the instruction fetch time) can either be estimated by high-level techniques such as basic block analysis, or obtained through rapid functional simulation of each thread. What needs to be captured are the data dependencies between instructions. For example, if one instruction writes to register *r3*, and then a later instruction reads from *r3*, we resolve this dependency by not issuing the latter before the former, as soon as this condition is identified. In this work, we use the second method of obtaining instruction fetch times, that of rapid functional simulation.

The following paragraphs describe the manner in which the messages of each instruction are assigned timestamps:

1. **Instruction fetch time.** This is estimated based on the fetch cycle of the last few instructions (only as many instructions as the fetch width may be fetched in one cycle), as well as the type of the last instruction. For example, no instructions may be fetched in a given cycle after a branch is encountered.
2. **Instruction issue time.** This is optimistically estimated as the greater of (1) the time at which it was fetched plus one cycle, and (2) when all its dependent instructions have completed execution.
3. **Instruction completion time.** This is the issue time plus the latency of the functional unit which this instruction uses. For instructions with memory operands, this translates to the time it takes to retrieve data from cache or memory.

In addition to inter-instruction dependencies within the same core, inter-core instruction dependencies exist. Even though it is straightforward to calculate the expected latency for a message to travel between two nodes from the

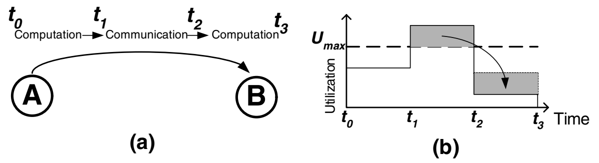


Figure 5: An example of inter-core dependency: (a) Node A performs internal computation, then sends a block of data to node B, which in turn performs its own internal computation. (b) Link utilization function for the link between nodes A and B; competing traffic from other nodes causes contention (area above U_{max}) between t_1 and t_2, which is propagated until after t_2 (dotted box).

distance and the pipeline depth of each router in the network, it is not possible to statically anticipate whether a particular message will be delayed in the network due to contention (and how many cycles it will be delayed). Therefore, it is important for our model to be able to handle such inter-core dependencies and hence model the closed-loop nature between the processor cores and the network fabric of CMPs. To illustrate this point, consider the scenario depicted in Figure 5. Here a computation block on node A generates a block of communication from node A to node B, which in turn feeds a computation block on node B. It is easy to calculate the zero-load time at which the communication will be complete, but it is possible that communication between other nodes will contend for the same link between node A and node B, thus reducing the available bandwidth for node A's communication to node B. Figure 5(b) shows the effect of contention in the network propagating some of the communication to the next time segment. Because the communication path overlaps with the path of the computation on node B which utilizes the communication, this propagation will in turn cause contention during the computation block on node B, delaying a number of the subsequent instructions. The capturing of this dynamic interaction between the processor computation and the network communication is a distinctive feature of our framework.

3.4 A Detailed Framework Walkthrough

We proceed with a more concrete example that complements the preceding description of our framework. Here, we present a walkthrough starting from the input: the assembly code which is given in Table 1, the (partial) ISA description given in Figure 4, and the architectural model of a processer core depicted as a network in Figure 3. We will show the way that the instructions are mapped onto this network, the generation of the injection rate functions, and the utilization function output for a single link, that of the integer ALU.

Input: assembly code. The example assembly code used here is a sequence of three simple repeated loops that begin with labels *blockA*, *blockB*, and *blockC* as Table 1 shows. Each loop repeats for one thousand instructions. The reasons for showing these repeated loops are two-fold: first, the injection rate functions that LUNA takes as inputs are described as piecewise constant functions, where each "piece"[5] is described by a pair of numbers, the time at which the segment begins and its (utilization) value; second, the use of a small code snippet enables us to feasibly present our walkthrough.

[5]Hereafter referred to as a "segment."

Table 1: Example instruction trace. Vertical ellipses denote the previous instructions looping repetitively.

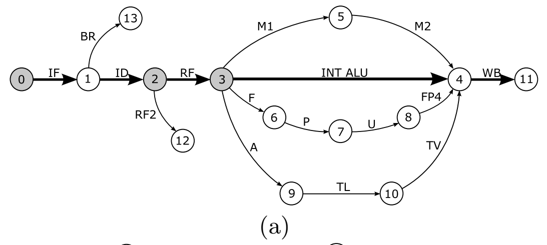
Label	Instr. no.	Instr.	Operands
blockA	1	addiu	R2 R2 8
	2	sw	R3 0(R2)
	3	j	blockA
	⋮	⋮	⋮
blockB	1001	addiu	R1 R1 8
	1002	mulh	R1 R5
	1003	addu	R4 R4 R1
	1004	j	blockB
	⋮	⋮	⋮
blockC	2001	add.s	f4 f4 f2
	2002	add.s	f8 f8 f6
	2003	j	blockC
	⋮	⋮	⋮

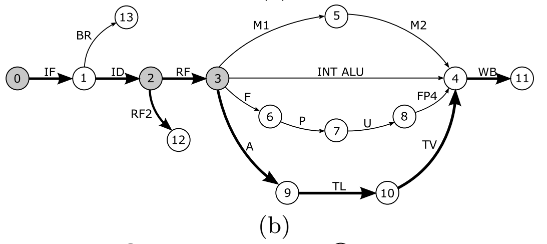
Table 2: Message injection rate functions for the example given by the code snippet in Table 1. Paths are given as a sequence of network nodes.

Message path	Message injection rate function
0 1 2	(0, 1.0) (1000, 1.0) (2000, 1.0)
2 3	(0, 0.33) (1000, 0.75) (2000, 0.67)
3 4 11	(0, 0.33) (1000, 0.50) (2000, 0.0)
2 12	(0, 0.33) (1000, 0.50) (2000, 0.67)
3 9 10 4 11	(0, 0.33) (1000, 0.0) (2000, 0.0)
3 6 7 8 4 11	(0, 0.0) (1000, 0.0) (2000, 0.67)
3 5 4 11	(0, 0.0) (1000, 0.25) (2000, 0.0)

Mapping instructions to messages in the network. The first instruction in Table 1 is `addiu`. At this point only the instruction itself is important and not the operands. An `addiu` lookup in the ISA table description (see Figure 4) reveals that the `addiu` instruction maps to three messages: one that reads from the instruction cache and decodes the instruction; a second that reads a register value; and a third that travels through the integer ALU, the result bus, and finally the commit stage, where the destination register is written. This mapping is depicted in Figure 6(a). Likewise, the mapping of the next two instructions, `sw` and `j`, are shown in Figures 6(b) and 6(c).

Calculating message injection rate functions. To generate the message injection rate functions, we parse the instructions one at a time. We keep a hash table where the keys are the paths (e.g. $0 \rightarrow 1 \rightarrow 2$), and the values are the number of times that a message has been read from the ISA description since the last segment. The values are reset at every sampling interval; in this case we choose the sampling interval to be one thousand instructions. At the end of each inteval, the value for each message is divided by the segment length. This is the average injection rate for the current message and for the current segment. As an example, consider the message from node 0 to node 1 to node 2 ($0 \rightarrow 1 \rightarrow 2$). Note that each instruction generates this message, and therefore, the number of times that this message occurs during the entire program is once per instruction. As





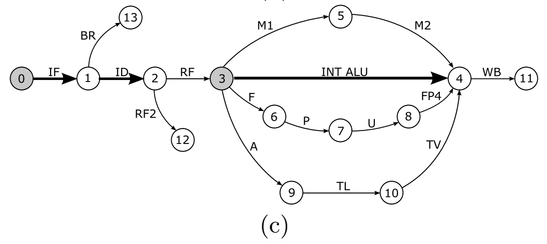


Figure 6: Three example instruction mappings: (a) `addiu`, (b) `sw`, and (c) `j`. Bold directed lines denote links upon which respective instructions are mapped. Filled circles at the beginning of directed arrows denote the beginnings of messages.

a consequence, the injection rate function for this message is 1.0 (or one message per instruction) for the entire length of the code. As another example, consider the message from node 3 to node 4 to node 11 ($3 \rightarrow 4 \rightarrow 11$). In this case, the only instructions that generate this message are `addiu`, `addu`, and `sw`. For the first one thousand instructions (*blockA*), one of the three instructions (`addiu`) generates this message (see the ISA description of Figure 4) and therefore the injection rate function is equal to 0.33 for the first segment. For the next one thousand instructions (*blockB*), two out of every four instructions generate this message, so the injection rate for the second segment is 0.50. Finally, for the last one thousand instructions (*blockC*), no messages are generated, so the injection rate for the last segment is 0.0. The rest of the message injection rate functions are generated in this way. Table 2 gives the injection rate functions for each of the messages encountered by translating the code given in Table 1.

Generating relative power profiles using LUNA. We make use of an example with a single link to show how we generate the relative power profile of the whole network. Consider the link between nodes 3 and 4 (the integer ALU). This link is only covered by one message: $3 \rightarrow 4 \rightarrow 11$. As a result, the relative power profile is simply given by the injection rate function of this message. If there were multiple messages traversing this link, then the relative power profile would be equal to the sum of the contributing injection rate functions of all such messages. Finally, if this sum is greater than the bandwidth (as specified in the architectural description) of a given link and at a given segment,

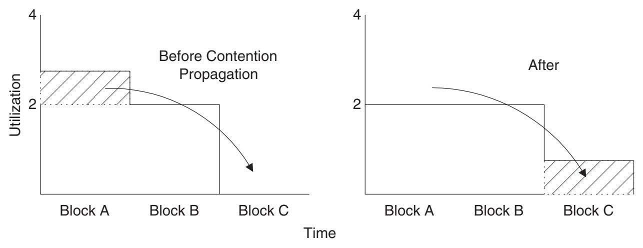


Figure 7: A simple contention propagation example.

then this contention needs to be propagated into the next segment(s). This model of propagation is incorporated into LUNA. As in the case of a single link, the same procedure described above is carried out on all links to generate the relative power profile of the entire network (processor). The result is simply the sum of all the individual link utilization functions.

As an example of how we employ LUNA to model resource contention in a processor, we consider the scenario illustrated in Figure 7. Let us suppose that the pipeline width (fetch, issue, decode, commit) in the network shown in Figure 6 is four instructions per cycle and that there are only two integer ALU units. In this case, the injection rate functions in Table 2 are multiplied by 4 (the fetch width). The injection rate function of the message $3 \rightarrow 4 \rightarrow 11$ multiplied by 4 is shown on the left side of Figure 7. Because there are only two integer ALUs, it is not possible to achieve a throughput of greater than two instructions per cycle through the execution stage. This utilization function is shown by the shaded area in Figure 7, which exceeds the maximum utilization of 2 (instructions per cycle). To reflect contention, this area is propagated to a later time frame. This area represents instructions which must stall because there are no appropriate execution units available. The resulting utilization function is shown in the right half side of Figure 7. This contention modeling and propagation is handled in LUNA in the same way network contention is handled.

3.5 Discussion

In the following subsections we discuss various features of modern out-of-order superscalar processors and how these features are incorporated into our analysis framework.

Branches and control dependencies. Our current implementation follows the branching behavior as observed from the functional simulation of the specific program and input data set. A more realistic implementation will have to handle control dependencies. This can be incorporated by flagging branch instructions with an estimated probability of being taken, which will then instruct the analysis to probabilistically inject the corresponding instruction/message (or not) into the processor/network. This probability can be gathered from compiler profiling (up to 85% branch prediction accuracy [17]) of representative input data sets to form a natural point of integration between the compiler and our analysis framework.

Leakage power. In our framework, leakage power is modeled by adding a constant energy cost to each link (resource). In the context of the network representation, the way this is done is as follows. First, the network is duplicated. That is, a duplicate copy of each link and node is created, with indices $i + N_l$ and $j + N_n$ representing this duplication, where i is the index of the current link, j is the index of the current node, N_l is the number of links, and N_n is the number of nodes. Each link is given a normalized bandwidth of 1.0. A new message is created for each link which *only* traverses one link, and is given an injection rate of 1.0 (*i.e.* maximum injection) for the entire time span. The energy cost of each of the links in the duplicated network is assigned to be the estimated leakage power for the respective resource. This ensures that the same leakage power will be added to each point in the profile of each resource. Note that this modification does not require knowledge of the program code (except for the code length) and thus does not lead to a run time increase in our framework.

Cache misses. The L2 (and subsequent levels of) cache is the most difficult resource to model because the L2 cache is only accessed if the L1 cache misses. In order to determine whether a particular L1 access is a hit or a miss, one must know the state of the cache as well as the value of the address being accessed. There are two approaches to this problem. The first solution is detailed profiling. Instead of our framework using as input just the assembly code from the compiler, additional information is conveyed to capture whether each memory access (including instructions) is expected to hit and if so, in which level of the cache it hits. Another approach, which is the one we take, assumes a pre-profiled miss rate for each cache. This approach probabilistically determines if a memory access hits and if so, in which cache level it hits. This compromises accuracy; however, as is evident in our validation, this loss in accuracy is not significant for high-level power analysis.

Level of pipeline detail. While our framework is flexible enough in allowing the user to define the level of pipeline detail, as expected there is a tradeoff between accuracy, practicality, and complexity. A high level of detail allows for higher accuracy, but at an earlier design stage, the necessary relative energy values per resource modeled may be difficult to estimate accurately. A low level of detail allows for more rapid analysis, as well as the ability to obtain reasonable relative energy estimates more easily, but may not provide a resultant relative power profile with enough variation and fine granularity to qualify its use. Here we model the major components of the Raw core at the same granularity as that of the capacitance estimates that we have, and we use the available information to its maximum potential. As an exception, consider the pipelined execution units depicted in Figure 3. Since we use a single estimate of the energy for each unit for each completion of an operation, we assign the cost of the first link in each execution unit's network path to be that single estimate, while the costs of the rest of the links are assigned as zero (no cost). Thus, we can simplify Figure 3 by collapsing the pipelined execution unit network paths into one stage (link) each. However, we choose to model it as we originally presented for two reasons: first, the mapping of the architecture to the network is more intuitive; and second, this allows for the easy extension of breaking the energy cost down by stage. In other words, if we had an estimate of the energy cost for each stage of the floating point ALU, we would instead assign non-zero weights to all four links *F, P, U, FP4*, in order to increase the accuracy of the model.

4. PUTTING IT ALL TOGETHER: HIGH-LEVEL CMP POWER ANALYSIS

With both processor cores and NoCs modeled as nodes and links in LUNA, combining the models into a comprehensive high-level CMP power analysis framework now becomes a relatively simple undertaking. First, the processor core's network representation is replicated for each node in the network.[6] Then, links are added to connect the communication fabric to the processor pipelines. For example, Figure 8 depicts the complete MIT Raw CMP represented by our analysis framework and highlights the way in which the NoC and cores are connected. The Raw chip is an on-chip multiprocessor with sixteen homogeneous cores. Each core consists of an eight-stage in-order single-issue pipeline with a single-stage integer ALU, a two-stage integer multiplier, and a four-stage floating point unit. Additionally, there is 32KB of data cache and 96KB of instruction cache per core. The core is mapped by LUNA onto nodes and links by following the same methodology described in Section 3. To facilitate communication between cores, the Raw CMP features four separate networks with each arranged as a two-dimensional mesh topology. Two of the networks are static and two are dynamic; the communication over the static networks is determined at compile-time, whereas the communication over the dynamic networks, such as in the case of cache misses, is determined at run-time. The communication fabrics are weaved into the core processor pipeline by allowing the network buffers (four per port per network) to be addressed in the same way as any other registers in the ISA. Because of this tight integration, it takes only three cycles for a data packet (i.e. a register value) to travel from one processor core to an adjacent core.

Referring again to Figure 8, we see that the processor core has two links connecting it to the communication network (the dashed circle in the upper-right): one from the communication fabric to the beginning of the execute stage, and one from the end of the execute stage to the communication network. This is the key in integrating the two components of the CMP together in our framework. While this scheme fits the Raw network interface well because the Raw network is tightly woven into the core pipeline, the same scheme can be used to model other interfaces as well. For example, if a cache-coherent shared-memory CMP is modeled, the network interface will be through the cache; the incoming link from the NoC would then connect to the node between the address and tag lookup stages. This represents a cache write when data is received by the processor. The outgoing link to the NoC remains the same, because its source node is the destination node for the cache (i.e. tag verify) link already.

While the integration of the processor cores' models into the communication fabric model in LUNA is straightforward, i.e. by constructing the graph as depicted in Figure 8, integrating the processor instructions (mapped as messages in the processor cores) with the actual NoC messages is more challenging. Instructions which have no dependencies outside the core are treated in the same way as in our uniprocessor model discussed previously. However, like many other NoCs, the Raw CMP uses a message-passing architecture

[6]Naturally, only homogeneous CMPs with identical processor cores can have their models replicated. Heterogeneous chips simply require distinct network representations for each distinct core or subsytem.

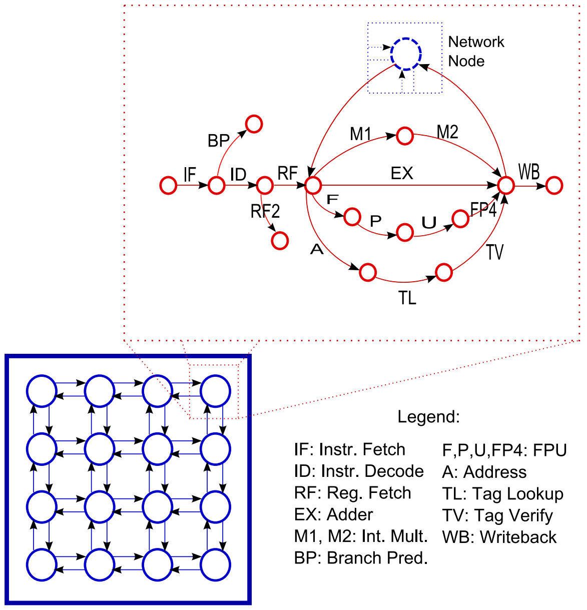


Figure 8: Raw CMP mapped to LUNA's nodes and links. The communication fabric is in the lower-left, and a processor core is depicted in the upper-right. This core is replicated at each node in the communication fabric. Note that the dashed node at the top of the core is actually a node in the communication fabric.

which means that there are explicit send and receive instructions at the programmer-level (e.g. using the C language). As mentioned earlier, at the assembly level, these sends and receives are carried out by reading from or writing to the registers which are actually the network buffers. Because stalls on one core or communication congestion can lead to stalls on another, it is necessary to capture this behavior in order to accurately create a temporal power profile. Therefore, a send or receive in our framework is modeled as a message whose source is a node in one core, but whose destination is a node in another core. A message that is stalled will not arrive at the destination core and thus will cause the injection of subsequent dependent messages to be stalled as well. The blocked messages then hold up resources in the processor core and network that in turn propagates congestion backwards to upstream messages and instructions. This allows intercore as well as intracore instruction dependencies to be modeled cleanly with LUNA's analysis of link utilization and contention.

4.1 Validation Results

We validate our methodology against the cycle-accurate Raw CMP BTL simulator. The Raw CMP BTL simulator [29] has been validated against actual silicon at a cycle-level granularity. At each cycle, the level of activity at each microarchitectural component is calculated by the BTL simulator and coupled with detailed capacitance values generated by the IBM Chipbench placement tool [19]. These values are low-level and accurate estimates which were obtained based on the final layout of the MIT Raw chip prior to tape-out. This allows the calculation of the power consumption of the Raw chip at the cycle level.

Our high-level analysis takes as input the instruction trace of all sixteen tiles of the Raw CMP. This trace was obtained from the output of the BTL simulator, then abstracted into 1000-cycle segments and fed into our framework (see Section 3.4), which then analyzes and returns a relative power profile for the entire CMP. To calculate the relative error, the cycle-level Raw power profile from the BTL simulator is sampled at the same frequency as our relative power profile, and both profiles are normalized to their respective means. Therefore, the relative error is calculated as the average absolute difference between the two profiles on a sample-by-sample basis:

$$Error = \frac{1}{N}\sum_{i=1}^{N} abs(P_{l_i}/P_{l_{avg}} - P_{r_i}/P_{r_{avg}}) \qquad (6)$$

where P_{l_i} is the ith sample of our relative power trace, $P_{l_{avg}}$ is the average value of our relative power profile across all samples, P_{r_i} is the ith sample of the Raw power trace, and $P_{r_{avg}}$ is the average value of the Raw power profile across all samples. The relative errors for a range of benchmarks that we ran on the Raw simulator are given in Table 3 while the temporal profiles of two benchmarks (`fft` and `gzip`) are shown in Figures 9 and 10.

While the average relative error is 9.1%, the most noteworthy trend is that half of the benchmarks exhibited less than 4% relative error, while the other half of the benchmarks exhibited greater than 13% relative error. The reason for this is that Equation 6 exaggerates the error percentage when there is more variation in the profiles, that is, fluctuation within the same profile. It is evident from Figure 9 that even though the profiles match well visually, upon close inspection, from approximately cycle 150,000 to the end of the trace, BTL's output varies much more than that of LUNA; this is a major source of the relative error measured. Similarly, in Figure 10, the profile of `gzip` is quite flat for both power analysis tools due to the fact that `gzip` performs the same operation on a continuous stream of data. Because there is limited variability, we consequently observe the lowest relative error. Our analysis framework also offers a significant speedup over the Raw BTL cycle-accurate simulator which is close to an order of magnitude slower.[7] Thus we see that our framework will be of use to compilers as an early-stage power analysis framework which offers good relative accuracy and a significant reduction in run time as compared to current high-level cycle-accurate power simulators.

4.2 Effect of High-Level Energy Estimation Error

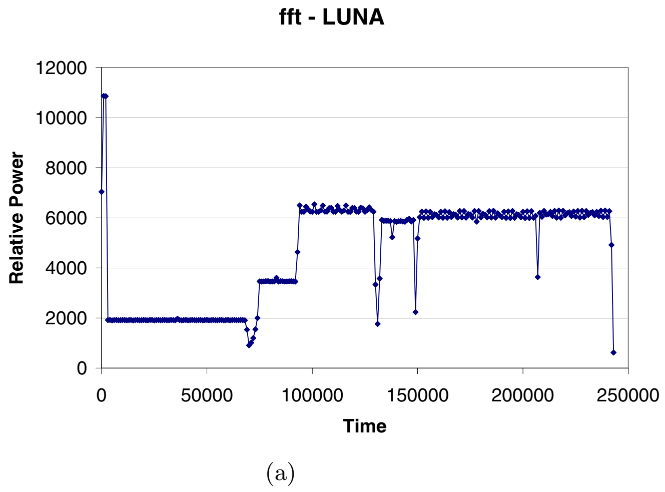
The previous sections discussed a number of methods for estimating the energy consumption of each processor and network resource by either using low-level capacitance calculations or higher-level area estimates. In detail, Section 4.1 demonstrated results using energy estimates based on the detailed low-level capacitance values of Raw. To further demonstrate the relative insensitiveness of our framework to energy estimation errors, here we randomly perturb the energy estimates for each resource, $E_{component_i}$, and investigate the effect on the accuracy of our approach.

Energy perturbation is carried out as follows: each CMP

[7]There is still room for future improvement as we did not perform any optimizations on our framework short of turning on the -O flag of gcc.

Table 3: Relative error and speedup of our framework as compared to the Raw CMP BTL simulator.

Benchmark	802_11a_enc	8b_10b_enc	fft	gzip	twolf	bzip2	avg
Relative error (%)	1.1	13.5	13.8	0.7	3.6	22	9.1
Speedup	9.1x	12.1x	7.4x	3.9x	4.1x	3.8x	6.7x



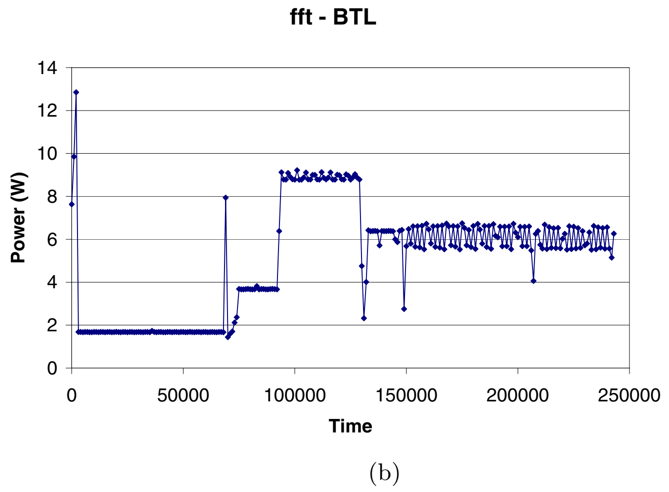
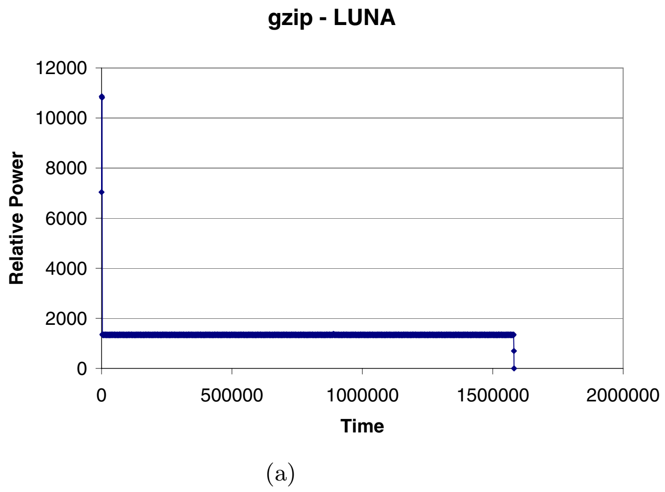


Figure 9: Power profiles for the `fft` benchmark running on the Raw CMP as captured by (a) LUNA and by (b) BTL.



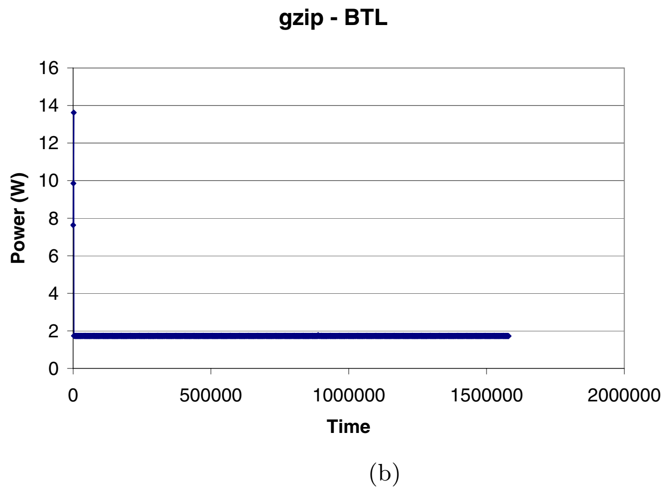


Figure 10: Power profiles for the `gzip` benchmark running on the Raw CMP as captured by (a) LUNA and (b) BTL.

resource energy cost $E_{component_i}$ (denoted with a one-hop link in our NoC abstraction model, see Section 3) now takes a random value $E = E_{component_i} \cdot P$, where P is a number between $\frac{1}{1+p_{max}}$ and $1+p_{max}$, i.e. P is evenly distributed in the $[\frac{1}{1+p_{max}}, 1+p_{max}]$ range. For example, with $p_{max} = 1.0$, the energy estimates randomly lie between 0.5 and 2X of their original values. Energy perturbation results for all six tested benchmarks are presented in Figure 11.

For each benchmark we ran four sets of experiments with different p_{max} values such that $p_{max} = \{0.1, 0.2, 0.5, 1.0\}$. We ran five experiments for each p_{max} for a total of twenty experiments per benchmark. The result for each experiment is a relative error value, as Table 3 demonstrates in the original case where no energy cost perturbations are applied, i.e. $p_{max} = 0$ and where the range $[\frac{1}{1+p_{max}}, 1+p_{max}]$ contracts to a single point and $P = 1$. It is noted that by simply averaging the results for each set of experiments for a given benchmark and p_{max} does not reveal much variation in average accuracy because each new set of energy estimates may decrease the resultant relative error value as well as increase it. Instead, here we plot the *range* between the minimum and maximum relative error estimates from each set of experiments in order to capture the power profile variations for the various benchmarks. The values in Figure 11 are normal-

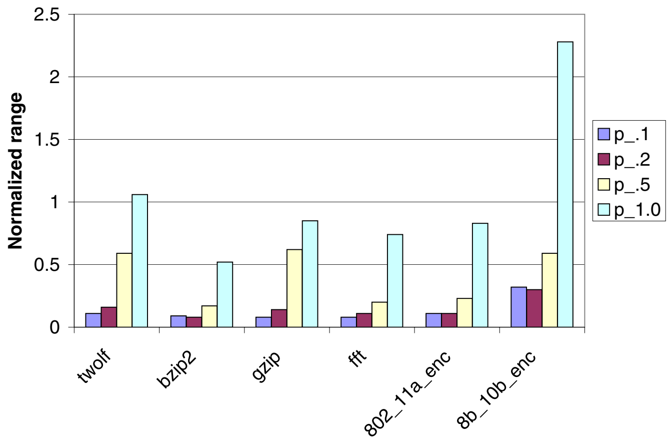


Figure 11: Normalized range of relative error as a function of perturbation magnitude (energy estimate inaccuracy). The legend gives p_{max} values.

ized to the values in Table 3, for each respective benchmark tested. For example, a variation of 1.06 for `twolf` and p_{max} = 1.0 means that the difference between the minimum and maximum calculated relative errors is 3.82% (3.6% $\times$ 1.06). The average error values across all benchmarks for p_{max} = 0.1 and p_{max} = 0.2 are 0.14 and 0.16 respectively. This indicates that energy estimates of up to 20% lead to less than 20% deviation in the calculated relative error. Thus, substituting higher-level energy or area estimates does not introduce large errors into our framework showing its high resilience to energy component resource perturbations.

5. RELATED WORK

There has been substantial prior work in the area of rapid power estimation for uniprocessor cores [16]. Here, due to space limitations, we present selected prior related work. These techniques, however, ignore the communication fabric's impact on overall power-performance in a multi-core chip. Power analysis of the processing elements of a CMP without regard to the communication fabric has two main disadvantages. First, a CMP is a closed-loop system, and therefore communication delays due to congestion or contention introduce dependencies between instruction flows of different cores which cannot be predicted by program analysis alone. Second, the power consumption of the network fabric comprises a significant percentage of total CMP power. For example, the four overlapping mesh networks consume 36% of the Raw CMP's power [14]. Tiwari et al. [38, 37] proposed fast software power analysis where the power consumed by each type of executed instruction is physically measured so a program's power consumption can be estimated by summing the average power consumption of single instructions or pairs of instructions at high accuracy, albeit only where the hardware platform is already available for measurement. Wattch [3], a library of architectural power models for uniprocessors, has been incorporated into cycle-accurate simulators such as SimpleScalar, achieving a fast simulation speed of about 400 million instructions per hour [32], as well as Simics, a full-system simulator [7, 31] (80 million instructions per hour, or roughly 1/5th the speed of SimpleScalar). A low-level power estimation framework is SimplePower [41], an execution-driven, cycle-accurate, RT-level power estimation tool based on SimpleScalar. Power simulation can be further accelerated through sampling, such as the use of SimPoints [30], where a small number of simulation points are chosen statistically to represent the complete execution of the program. While this technique does not increase the simulation speed, per se, in terms of instructions per hour, it reduces the time to arrive at overall performance estimates by approximately three orders of magnitude. However, our goal here is to provide a power profile as a function of time rather than average power, so the SimPoints approach is not suitable for our framework. Recent work by Isci et al. [13] collects and analyzes power phases in running applications. These phase characterizations can summarize application behavior with representative execution regions, thus selectively fast-forwarding simulations. However the technique requires a real-system measurement framework which means that the system under test needs to be available in silicon.

More recent research has explored power estimation for multi-core chips, looking at both computation and communication cores. Meyer et al. [18] take a higher-level approach to system-level simulation, exploring thread or fragment-level rather than instruction-level power-performance simulation. Similarly to the work of Tiwari et al. [38], Meyer et al. statically estimate energy costs per fragment rather than per instruction. Despite being rapid, this approach estimates aggregate power estimates instead of providing a temporal profile. The publicly-released multiprocessor simulator, RSIM [21], has been extended with power models from Orion [40] for bus-based multi-core chips [27] (roughly 40 million instructions per hour). Similarly, Orion has been integrated into the Liberty Simulation Environment [39] to construct a cycle-accurate interconnection network power simulator and can be extended for power modeling of the entire CMP (between 2 million and 130 million instructions per hour, for 16 and 1 cores simulated, respectively). Other multiprocessor power simulators such as that of Raw [14, 34] are specific to their chip architectures and offer little flexibility in investigating other design alternatives. In general, these detailed full-system simulators are very useful for architectural research, but the slow simulation speed renders them unsuitable for incorporating them within a compiler flow.

6. CONCLUSIONS

High-level power analysis that enables rapid, early-stage power estimation with good relative accuracy and which cannot afford lengthy simulations in order to make power-aware decisions, is critically needed, particularly within the compiler flow. Additionally, this problem escalates with the advent of multi-core chips that require prohibitive simulation lengths with cycle-level architectural power simulators: in these simulators, the simulation time scales super-linearly as a function of the number of cores under consideration.

In this paper we proposed a high-level power analysis framework that synergistically considers both the processor cores and the communication fabric in a multi-core chip. This technique maps processor pipelines as networks, thereby leveraging an existing high-level network power analysis framework that has been shown to be suitable for incorporation into the compiler flow. In this paper we demonstrated its effectiveness by analyzing the entire MIT Raw CMP, realizing 7X speedup on average versus the cycle-accurate Raw BTL simulator, while maintaining a relative accuracy of 9.1% on average.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments for this and future work. We would also like to acknowledge the support for this work: the MARCO Gigascale Systems Research Center, NSF grant CNS-0509402, and the Intel Corporation.

7. REFERENCES

[1] ARM Integrated Multiprocessor Core, 2006. Available [online]: http://www.arm.com.

[2] L. Benini and G. De Micheli. *Powering Networks on Chip*. In Proc. of the International Symposium on System Synthesis, pp. 33-38, Oct. 2001.

[3] D. Brooks et al. *Wattch: A Framework for Architectural-Level Power Analysis and Optimizations*. In Proc. of the International Symposium on Computer Architecture, pp. 83–94, June 2000.

[4] M. Burtscher and I. Ganusov. *Automatic Synthesis of High-Speed Processor Simulators*. In Proc. of the International Symposium on Microarchitecture, pp. 55–66, Dec. 2004.

[5] G. Chen et al. *Compiler-Directed Channel Allocation for Saving Power in on-Chip Networks*. In Proc. of the Symposium on Principles of Programming Languages, pp. 194–205, Jan. 2006.

[6] G. Chen et al. *Reducing NoC Energy Consumption Through Compiler-Directed Channel Voltage Scaling*. In Proc. of the Conference on Programming Language Design and Implementation, pp. 193–203, June 2006.

[7] J. W. Chen et al. *SimWattch: An Approach to Integrate Complete-System with User-Level Performance/Power Simulators*. In Proc. of the International Symposium on Performance Analysis of Systems and Software, pp. 1–10, March 2003.

[8] W. J. Dally and B. Towles. *Route Packets, Not Wires: On-Chip Interconnection Networks*. In Proc. of the Design Automation Conference, pp. 684–689, June 2001.

[9] L. Eeckhout et al. *Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox*. IEEE Micro, Vol. 23, No. 5, pp. 26–38, Sept.-Oct. 2003.

[10] N. Eisley and L.-S. Peh. *High-Level Power Analysis for on-Chip Networks*. In Proc. of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp. 104–115, Sept. 2004, LUNA: Available [online] http://www.princeton.edu/∼eisley/LUNA.html.

[11] Gigascale Systems Research Center (GSRC), 2006. Available [online] http://www.gigascale.org/roadmap/.

[12] C.-T. Hsieh et al. *Profile-Driven Program Synthesis for Evaluation of System Power Dissipation*. In Proc. of the Design Automation Conference, pp. 576–581, June 1997.

[13] C. Isci and M. Martonosi. *Phase Characterization for Power: Evaluating Control-Flow-Based and Event-Counter-Based Techniques*. In Proc. of the International Symposium on High Performance Computer Architecture, pp. 121–132, Feb. 2006.

[14] J. S. Kim et al. *Energy Characterization of a Tiled Architecture Processor with on-Chip Networks*. In Proc. of the International Symposium on Low Power Electronics and Design, pp. 424–427, Aug. 2003.

[15] P. Kongetira et al. *Niagara: A 32-Way Multithreaded Sparc Processor*. IEEE Micro, Vol. 25, No. 2, pp. 21-29, March/April 2005.

[16] E. Macii et al. *High-Level Power Modeling, Estimation, and Optimization*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 17, No. 11, Nov. 1998.

[17] S. McFarling and J. Hennessy. *Reducing the Cost of Branches*. In Proc. of the International Symposium on Computer Architecture, pp. 396–403, June, 1986.

[18] B. H. Meyer at. al. *Power-Performance Simulation and Design Strategies for Single-Chip Heterogeneous Multiprocessors*. IEEE Transactions on Computers, Vol. 54, No. 6, June 2005.

[19] MIT Raw Team, personal communication, 2006.

[20] J. Oliver et al. *Synchroscalar: A Multiple Clock Domain, Power-Aware, Tile-Based Embedded Processor*. In Proc. of the International Symposium on Computer Architecture, pp. 150–161, June 2004.

[21] V. S. Pai et al. *RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors*. In Proc. of the International Symposium on High Performance Computer Architecture, pp. 72–83, Feb. 1997.

[22] M. Pedram and Q. Wu. *Design Considerations for Battery-Powered Electronics*. In Proc. of the Design Automation Conference, pp. 861–866, June 1999.

[23] D. Penry et al. *Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-Processors*. In Proc. of International Symposium on High Performance Computer Architecture, pp. 29–40, Feb. 2006.

[24] D. Pham et al. *The Design and Implementation of a First-Generation Cell Processor*. In Proc. of the International Solid-State Circuits Conference, pp. 184–185, March 2005.

[25] PoPNet Simulator, 2006. Available [online] http://www.princeton.edu/∼lshang/popnet.html.

[26] K. Sankaralingam et al. *Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture*. In Proc. of the International Symposium on Computer Architecture, pp. 422–433, June 2003.

[27] R. Sasanka et al. *The Energy Efficiency of CMP vs. SMT for Multimedia Workloads*. In Proc. of the International Conference on Supercomputing, pp. 196–206, June 2004.

[28] L. Shang et al. *PowerHerd: Dynamically Satisfying Peak Power Constraints in Interconnection Networks*. In Proc. of the International Conference on Supercomputing, pp. 98–108 , June 2003.

[29] L. Shang et al. *Thermal Modeling, Characterization, and Management of On-Chip Networks*. In Proc. of the International Symposium on Microarchitecture, pp. 67–78, Dec. 2004.

[30] T. Sherwood et al. *Automatically Characterizing Large Scale Program Behavior*. ACM SIGPLAN Notices, Vol. 37 , No. 10, pp. 45–57, Oct. 2002.

[31] Simics, 2006. Available [online]: www.simics.net.

[32] SimpleScalar LLC, 2006. Available [online]: http://www.simplescalar.com.

[33] V. Soteriou et al. *Software-Directed Power-Aware Interconnection Networks*. In Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 274–285, Sept. 2005.

[34] M. B. Taylor et al. *Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams*. In Proc. of the International Symposium on Computer Architecture, pp. 2–13, June 2004.

[35] M. B. Taylor et al. *Scalar Operand Networks*. IEEE Transactions on Parallel and Distributed Systems, Vol. 16, No. 2, pp. 145–162, Feb. 2005.

[36] The Standard Performance Evaluation Corporation, 2006. Available [online]: http://www.spec.org.

[37] Vivek Tiwari et al. *Instruction Level Power Analysis and Optimization of Software*. Journal of VLSI Signal Processing, Vol. 13 Issues 2–3, pp. 223–238, Aug.-Sep. 1996.

[38] Vivek Tiwari et al. *Power Analysis of Embedded Software: A First Step towards Software Power Minimization*. In Proc. of the 1994 International Conference on Computer-Aided Design, pp. 384–390, Aug. 1994.

[39] M. Vachharajani et al. *Microarchitectural Exploration with Liberty*. In Proc. of the International Symposium on Microarchitecture, pp. 271–282, Nov. 2002.

[40] H. Wang et al. *Orion: A Power-Performance Simulator for Interconnection Networks*. In Proc. of the International Symposium on Microarchitecture, pp. 294–305, Nov. 2002.

[41] W. Ye et al. *The Design and Use of SimplePower: A Cycle Accurate Energy Estimation Tool*. In Proc. of the Design Automation Conference, pp. 340–345, June 2000.

[42] K. C. Yeager. *The MIPS R10000 Superscalar Microprocessor*. IEEE Micro, Vol. 16, No. 2, pp. 28–40, April 1996.