Micro-Architecture Independent Analytical Processor Performance and Power Modeling

Sam Van den Steen*, Sander De Pestel*, Moncef Mechri+,
Stijn Eyerman*, Trevor Carlson+, David Black-Schaffer+, Erik Hagersten+ and Lieven Eeckhout*

*Department of Electronics and Information Systems, Ghent University, Belgium
+Department of Information Technology, Uppsala University, Sweden

***Abstract*—Optimizing processors for specific application(s) can substantially improve energy-efficiency. With the end of Dennard scaling, and the corresponding reduction in energy-efficiency gains from technology scaling, such approaches may become increasingly important. However, designing application-specific processors require fast design space exploration tools to optimize for the targeted application(s). Analytical models can be a good fit for such design space exploration as they provide fast performance estimations and insight into the interaction between an application's characteristics and the micro-architecture of a processor.**

Unfortunately, current analytical models require some micro-architecture dependent inputs, such as cache miss rates, branch miss rates and memory-level parallelism. This requires profiling the applications for each cache and branch predictor configuration, which is far more time-consuming than evaluating the actual performance models. In this work we present a *micro-architecture independent* profiler and associated analytical models that allow us to produce performance *and* power estimates across a large design space almost instantaneously.

We show that using a micro-architecture independent profile leads to a speedup of 25× for our evaluated design space, compared to an analytical model that uses micro-architecture dependent profiles. Over a large design space, the model has a 13% error for performance and a 7% error for power, compared to cycle-level simulation. The model is able to accurately determine the optimal processor configuration for different applications under power or performance constraints, and it can provide insight into performance through cycle stacks.

I. Introduction

Over the past few decades, computing energy-efficient gains came primarily from improvements in process technology. Every new process generation gave smaller transistors, which resulted in faster switching speeds *and* also lower energy, due to proportional voltage reductions. Taken together, this resulted in an almost constant power density, as Dennard predicted [1]. However, threshold voltage scaling limitations and an increasing fraction of leakage power have ended this trend. Future process generations are not expected to deliver significant energy-efficiency by themselves. This change places the burden of improving energy-efficiency in the hands of the architects, who have to figure out how to use the transistors more efficiently.

One way of improving energy-efficiency is to design application-specific processor cores [2]. Application-specific cores are tailored to specific application(s), by removing or reducing all components that are not used, or under-used, by the application(s) (e.g., smaller caches or a narrower pipeline), and/or enlarging and adding components that benefit the application (e.g., accelerators). Embedded processors are a typical use case for application-specific processors, because they execute a limited set of applications and can be tightly optimized. However, general-purpose processors can also benefit from application-specific processor design by only turning on functionality when it benefits the current application(s). This is particularly intriguing with the advent of dark silicon [3], which states that only a limited area of the chip can be powered up at a given time. To make effective use of such a chip, we can build a range of application-specific sub-processors, and only activate the one(s) tailored to the current workload, thereby achieving better efficiency.

However, designing optimal application-specific processors is challenging due to the huge design space. The architect has to determine the optimal pipeline depth and width, the sizes of the internal buffers (reorder buffer, load-store queue), the sizes of the caches and the number of levels, the memory bandwidth, etc. All of these parameters have an impact on performance and power consumption, and all are dependent on the characteristics of the application(s). This means that the detailed design process needs to be repeated for each application. To enable such designs, we need a tool that can quickly evaluate large design spaces to find interesting regions that can then possibly be explored with detailed simulations.

Analytical performance models are an excellent fit for pruning large design spaces because they use analytical formulas to produce performance estimates based on application characteristics enormously faster than simulation. However, current analytical models require inputs that depend on both the application and the micro-architecture (branch miss rates, cache miss rates, and memory-level parallelism). The micro-architecturally *dependent* applicaton inputs are a function of both the application and the underlying architecture, and must be profiled for every configuration of the cache and the branch predictor, meaning that an application needs to be profiled multiple times. Because profiling is the most time-consuming step of using analytical models, the number of profiling steps needs to be limited, and should be ideally done only once per application. In this paper, we present an analytical performance and power model based on purely micro-architecture *independent* application profiles, enabling the evaluation of a full design space with only one profiling step.

In particular, we make the following contributions:

- We discuss the changes to the original interval model [4], which was developed to model an Alpha processor, in order to accurately model contemporary Intel processors.

- We incorporate previously proposed micro-architecture independent models to estimate cache miss rates and branch miss rates, and evaluated the impact on the accuracy of the model.
- We propose a new model to estimate memory-level parallelism (MLP) based on a micro-architecture independent application profile.
- We propose a mechanistic power model to quickly estimate the power and energy consumption of a processor.
- We implemented a fast Pin-based profiler that records the required profile.
- We show that the resulting model is accurate and fast enough to enable the efficient exploration of large design spaces.

We will first cover prior work on modeling processors for design space exploration (Section II), and then introduce the extensions to interval modeling to enable the use of micro-architecturally independent profiles to model contemporary processors, branch predictors, arbitrary memory hierarchies, MLP, and power (Section III). To analyze the effectiveness of these enhancements, we compare the results from the new micro-architecturally independent interval model to detailed simulation for both accuracy and speed (Section IV), and explore how we can use these new fast models for broad design space exploration (Section V).

II. Related Work

Analytical Models: Early computer architecture research often used analytical models to evaluate their proposal. However, as designs became more complex, and available computing power increased, researchers switched to detailed simulations. Yet in the last decade this trend has begun to reverse as the community has realized that detailed simulation is becoming prohibitively slow with increasing parallelism, and, that while detailed, it provides little by the way of insight. These trends have led to a resurgence in the use of analytical models as a means to provide fast predictions and insight.

Empirical Models: One approach uses empirical models that are automatically built from a training set, e.g., through regression [5] or artificial neural networks [6]. These models are based on the premise that current processor micro-architectures are too complex to model, but that through machine learning we can calibrate a generic model to faithfully reproduce their behavior. Empirical models are easy to build and can be fairly accurate, but they need a non-negligible training set, requiring many slow simulations. Furthermore, they can suffer from overfitting or limited generalization if the training set is not diverse enough *and need to be built for every (set of) workload(s).* Such models rarely provide much insight into the mechanisms of a processor and why a certain application has a given performance on a given configuration.

Mechanistic Models: Another approach is mechanistic modeling, which builds on simplifying assumptions and first-order effects, observed by studying the flow of instructions through the processor pipeline. Such mechanistic models are generally less accurate than empirical models, and do not model the whole processor in detail. However, while mechanistic models have limited detail, they do reveal how the program interacts with the micro-architecture through mathematical formulas upon which they are built. This underlying simplicity allows them to provide more insight in the performance bottlenecks of a program or a processor, e.g., by building CPI stacks [7] or by analytically providing sensitivities to various parameters. A first approach to model a superscalar out-of-order processor with mechanistic models was made by Karkhanis and Smith [8], and was further refined by Eyerman et al. [4] into today's interval processor model. Chen and Aamodt [9] extended this work by adding prefetching and a more accurate memory model.

An intermediate approach is to use an intuitive mathematical model, where some of the parameters are determined by fitting results of a training set. This approach tries to combine the accuracy of empirical models with the insightfulness of mechanistic models. Examples include Hartstein and Puzack [10] model of pipeline depth and width, and the mechanistic-empirical model by Eyerman and Eeckhout [11] to build CPI stacks using existing hardware.

Mechanistic models consist of two phases: application profiling and performance estimation. Profiling is usually the most time-consuming step, as the instructions of the application need to be analyzed to obtain the application characteristics required by the mathematical model. One drawback of current mechanistic models is that some characteristics are obtained by simulating parts of the processor (micro-architecturally dependent inputs), such as the cache and the branch predictor. Although these simulations are faster than fully simulating the processor, they need to be redone for every configuration of the cache hierarchy and the branch predictor to be simulated. We tackle this problem in this paper by proposing a single micro-architecture independent profiling step that enables estimating performance and power consumption of a large processor design space.

Interval Simulation: The interval simulation technique [12] combines the profiling and modeling into a single step, thereby enabling fine-grained performance estimations. Interval simulation models the timing of individual instructions, such as memory operations, enabling the fast and accurate simulation of a multi-core processor with shared memory components. The difference between interval simulation and the interval model is that interval simulation needs to be redone if the configuration of the core changes, whereas the interval model can estimate performance for a large range of configurations. On the other hand, the interval model does not provide timings of individual instructions, complicating the modeling of interference in shared memory components, such as a shared cache.

Power Modeling: Several models to estimate the power consumption of a processor core have been developed. Some of them are tightly integrated with a performance simulator [13] or require activity factors that are generated using detailed simulation [14]. Other models estimate power consumption in real time by making use of performance counters [15]. To the best of our knowledge, there exist no power models that estimate the power consumption of a large range of processor core configurations using an analytical model. In this paper,

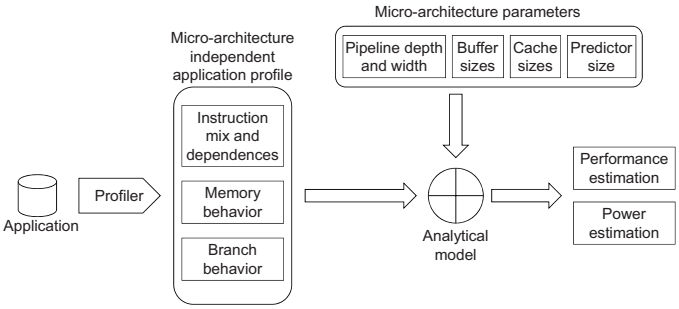

Fig. 1. Overview of the analytical performance and power model using micro-architecture independent application profiling.

we propose such a model by estimating activity factors based on the outputs of the interval model.

ISA-independent profiling: Shao and Brooks [16] propose to characterize workloads independent of the ISA (Instruction Set Architecture). They profile memory, branch and opcode behavior of an application using the intermediate representation of an application instead of the binary. Following up on this work, they propose using the ISA-independent profiles for speeding up the design of accelerators [17]. They achieve good speedups with minimal loss in accuracy for both performance and power estimations. Moving to an ISA-independent model to predict performance and power usage by leveraging the proposed techniques in these papers can be an extension to the models we propose.

III. Micro-Architecture Independent Model

A. Overview

To enable fast design space exploration for application-specific processors, we propose an analytical performance model based on a micro-architecture independent profile. This means that the application needs to be profiled only once, after which instantaneous performance and power estimates for a large design space can be made, including different sizes of caches and branch predictors. Figure 1 gives an overview of the method. The profiling step on the left needs to be done only once, obtaining an *application-dependent* but *micro-architecture independent* profile. The model itself consists of analytical formulas that take the application profile and micro-architectural parameters, and provide an estimation of the performance and power consumption. The model can be applied multiple times to obtain results for a large design space, without re-profiling the application.

The model is based on the interval model [4], a mechanistic performance model for out-of-order superscalar processors. To estimate performance, the original interval model required a few micro-architecture dependent inputs: the number of misses in each level of cache, the number of branch predictor misses and the amount of memory-level parallelism (MLP). These inputs are a complex function of the behavior of the application and the organization of the caches and predictors, and were obtained by simulation. In this paper, we model them in a micro-architecture independent manner, by profiling the application only once and using cache and branch predictor models to estimate the number of misses. Furthermore, we add power modeling to evaluate energy consumption, to allow the designer to balance performance and power in the design space exploration.

B. Modeling x86 processors

The interval model was originally validated against an Alpha architecture simulated on SimpleScalar. To make this approach more applicable today, we have updated the model to be able to handle a range of modern x86 processors.

The overall model to estimate the number of cycles C uses the number of instructions N, the effective dispatch rate D_{eff} (see later), the number of branch mispredictions m_{bpred}, the branch resolution time c_{res}, the frontend pipeline depth c_{fe}, the number of instruction fetch misses in each level i of the cache m_{ILi}, the access latency to each cache level c_{Li}, the number of last-level cache misses m_{LLC}, the memory access time c_{mem}, the memory bus transfer and waiting time c_{bus}, and the amount of memory-level parallelism MLP.

$$C = \frac{N}{D_{eff}} + m_{bpred}(c_{res} + c_{fe}) + \sum_i m_{ILi} c_{Li+1} + \frac{m_{LLC}(c_{mem}+c_{bus})}{MLP} \quad (1)$$

This model is slightly different from the model in [4], to handle the differences between x86 and Alpha. The first important difference between them is that x86 is a CISC architecture, while Alpha is a RISC architecture. However, the internal processor pipeline in a modern x86 processor also uses a RISC instruction set, which is implemented by transforming x86 instruction into micro-operations. Therefore, we first compute the sequence of micro-ops from the x86 code. As a result, the N in Equation 1 is equal to the number of micro-ops, and not the number of instructions. On average, there are 1.2 micro-ops per x86 instruction.

The number of micro-ops is divided by the effective dispatch rate in the absence of misses. This equals the minimum amount of cycles it takes to execute a program and is called the base component or base performance. For the Alpha model, this dispatch rate is set to the dispatch width D of the processor, assuming the reorder buffer (ROB) is large enough to sustain an IPC of D (balanced design). However, we found that since the x86 architecture offers fewer architectural registers, more spill code is generated, and the dependence paths tend to be longer. This causes the dispatch rate to be lower than the dispatch width. Therefore, we define the effective dispatch rate D_{eff}, which is calculated using Little's law as follows:

$$D_{eff} = \min\left(D, \frac{ROB_{eff}}{lat \times K(ROB_{eff})}\right) \quad (2)$$

$K(ROB_{eff})$ is the critical dependence path in a window of instructions equal to the effective ROB size (see later, Formula 4 shows how to calculate ROB_{eff}) and lat is the average instruction latency, including short (L1 and L2) data cache misses.

The effective dispatch rate can also be limited by the number of functional units. For example, if there is only one load unit, the performance of a sequence of instructions with many loads will be limited because only one load can be executed per cycle. In general, if there are N_i instructions of type i, and U_i functional units of that type, then it takes at least $\frac{N_i}{U_i}$ cycles to execute this code. For non-pipelined units with

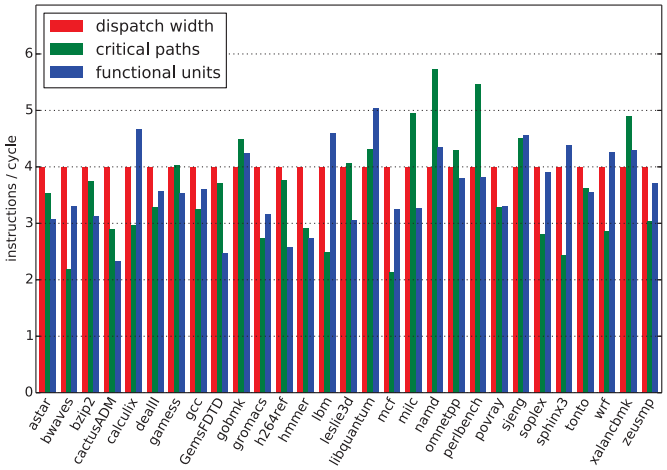

Fig. 2. The three factors limiting the effective dispatch rate: the dispatch width, the critical path via inter-instruction dependences, and the number of functional units of a particular type.

latency lat_i, the minimal execution time is $\frac{\text{N}_\text{i} \times \text{lat}_\text{i}}{\text{U}_\text{i}}$. Therefore, we can rewrite the effective dispatch rate from Equation 2 as

$$\text{D}_\text{eff} = \min\left(\text{D}, \frac{\text{ROB}_\text{eff}}{\text{lat} \times \text{K}(\text{ROB}_\text{eff})}, \frac{\text{N} \times \text{U}_\text{i}}{\text{N}_\text{i}}, \frac{\text{N} \times \text{U}_\text{j}}{\text{N}_\text{j} \times \text{lat}_\text{j}}\right) \quad (3)$$

where i ranges over all types of pipelined functional units, and j over all non-pipelined functional units. Figure 2 visualizes the factors that limit D_eff, following Formula 3 – note that we do not make a distinction between pipelined and non-pipelined functional units (terms 3 and 4 in Formula 3). The most limiting factor for the base performance of each benchmark is represented by the lowest bar. For some benchmarks, the rightmost bar is lowest, which means that the effective dispatch rate is limited by the number of available functional units. We find this to be mainly a result of a high fraction of loads, due to spill code. For other benchmarks however, the middle bar is the lowest, which implies that inter-instruction dependences limit the effective dispatch rate. This typically occurs when the critical path is so long that it fills up the ROB. If neither of these factors limit the effective dispatch rate, the dispatch rate will be equal to the dispatch width, in which case the leftmost bar is the lowest.

We also model the impact of a limited number of reservation stations or issue queue (IQ) entries. The issue queue is usually smaller than the ROB because it contains a subset of the instructions in the ROB, i.e., the not yet executed instructions, and because it is a very power-hungry and time-critical content-addressable memory (CAM) structure. However, having too few issue queue entries can stall the processor before the ROB is fully occupied, meaning that we cannot exploit the full ROB size to extract instruction-level parallelism. To model this, we use an approach similar to Karkhanis and Smith [18]. They reason that the occupation of the issue queue divided by the occupation of the ROB equals the average dependence path length divided by the critical dependence path length. If the issue queue size is smaller than the ROB size times this fraction, then the processor stalls on a full issue queue before the ROB is fully occupied. This is calculated as:

$$\text{ROB}_\text{eff} = \min\left(\text{ROB}, \text{IQ}\frac{\text{lat}_\text{K} \times \text{K}(\text{ROB})}{\text{lat}_\text{A} \times \text{A}(\text{ROB})}\right) \quad (4)$$

with IQ the issue queue size (or the number of reservation stations), A(ROB) the average dependence path length over a window of ROB instructions, and lat_K and lat_A the average instruction latency along the critical path and the average path, respectively. We added the latter latencies, because we found a large difference between the instruction mix along the critical path and along the average path. Ignoring this difference results in a factor of 2, leading to a too large penalty if the issue queue is smaller than half the ROB size. Adding these latencies leads to a factor of 3, which is close to what Karkhanis found, and results in a better accuracy.

The occupancy in the load-store queue (LSQ) can be modeled using a similar approach. However, for the SPEC CPU2006 benchmarks and core configurations in our setup, we noticed that dependences, the average instruction mix and IQ size are more restrictive than the LSQ size.

The second term in Equation 1 is the branch misprediction component. It is calculated by the number of branch misses times the sum of the branch resolution time c_res and the front-end pipeline depth c_fe (fetch, decode and rename stages). In the Alpha model, a mispredicted branch is most often the last correct-path instruction to execute [19], so the branch misprediction penalty is defined as the critical path length. For x86, we found that a mispredicted branch is most often not on the critical path, because the critical path mainly consists of a chain of memory operations. Therefore, it is more accurate to model the branch resolution time as the average dependence path length (instead of the critical path), multiplied by the average instruction latency.

The third term in Equation 1 is the instruction cache miss component. Similar to the Alpha model, it is calculated as the number of misses in each level i multiplied by the access time to the next cache level $\text{i} + 1$.

The last term is the last-level data cache miss component, which is the number of misses times the memory access time, divided by the memory level parallelism (MLP), which is the average number of overlapping misses. c_bus is the cycles spent using the memory bus or waiting for the memory bus if it is occupied.

Without these changes, i.e., applying the Alpha model unchanged, the average error of modeling the performance of an Intel Nehalem-like processor was 20% with 15 out of 29 benchmarks having an error larger than 20%. With the changes described above, this is reduced to an average 11% error, and only 6 benchmarks with an error of more than 20%. Note that this evaluation is still done with the micro-architecture dependent inputs, i.e., the miss rates and MLP were obtained using simulation. The next sections describe how we profile memory and branch behavior in a micro-archictecture independent way, and how we estimate the miss rates and the MLP.

C. Branch misprediction rate modeling

To obtain the number of branch mispredictions, we measure the branch entropy of an application [20]. Branch entropy

quantifies how predictable branches are: an entropy of 0 means that branches are perfectly predictable, while an entropy of 1 indicates random branch behavior. We apply the technique described in [20] to create models that estimate branch miss rates from branch entropy. In practice, we profile the outcome of all conditional branches, record history tables, and calculate local, global and tournament branch entropy for different history lengths. For estimating the branch miss rate of a specific branch predictor in the performance model, we use a linear model that relates branch entropy to miss rate. To find this linear relationship, we use a set of (micro)benchmarks for which we both simulate all types of branch predictors and measure the entropy. The linear model for each type of predictor is then constructed using a least-squares fit on the results of the training set. Once the model is constructed, we do not need to perform any branch predictor simulation, and we can instantly obtain the branch miss rate for every new application, after profiling its branch entropy.

The estimated branch miss rate is used directly in Equation 1 as m_{bpred}, but is also used to calculate the branch resolution time c_{res}. We assume that the average number of instructions between two branch misses equals $\frac{N}{m_{bpred}}$, and we use the leaky-bucket method described in [4] to obtain the average number of instructions in the ROB when a branch miss occurs. As described above, we use the average dependence path length on a window containing this many instructions to obtain the branch resolution time, instead of the critical dependence path.

D. Cache miss rate modeling

In order to model caches, we use the StatStack statistical cache model [21], which provides the miss ratio for LRU caches of arbitrary size.

StatStack uses reuse distances (the number of memory references between two accesses to the same cache line) to compute cache behavior. Note that reuse distances count total memory references between accesses and not unique references, which makes them far cheaper to collect than stack distances. The reuse distances for an application are used to build a histogram of an application's reuse behavior, which is then transformed into a stack distance distribution. The stack distance distribution gives the number of unique cache lines accessed between two accesses to the same cache line, and, hence the miss ratio for an LRU cache by counting the number of accesses for a stack distance greater than the cache size.

StatStack reduces profiling overhead by collecting only a sample of the reuse distances in an application. Berg et al. [22] showed that it is possible to profile an application to get its reuse distance distribution with a very low overhead using hardware performance counters. This approach has been further optimized by Sembrant et al. [23]. In our current implementation we use Pin to collect this profile.

For the interval model, we have extended StatStack to differentiate between load misses and store misses, because store misses do not have a performance impact in the interval model, but they contribute to the power consumption of the cache. To estimate the miss ratios at each level of the cache hierarchy, we model each cache level independently using StatStack, which implicitly assumes inclusivity. Our evaluation shows that this provides good accuracy across a standard Nehalem-like three-level cache.

Instruction cache behavior is similarly modeled by the reuse distance distribution of the instruction address stream.

E. MLP modeling

Memory-level parallelism or MLP is defined as the number of last-level cache (LLC) misses that can be handled in parallel. Multiple misses that concurrently access main memory overlap, and therefore see a penalty that is equal to that of a single miss, explaining the division by the MLP in Equation 1. Two or more LLC misses can only be handled in parallel if they are both in-flight, i.e., together in the reorder buffer, and if they are independent of each other. This means that MLP depends on the number and position of the misses, dependences between misses, and the reorder buffer size, which makes it a challenge to model this in a micro-architecture independent way.

A naive approach is to assume that all misses are uniformly distributed across the application, and that there are no dependences. MLP then equals the number of misses times the ROB size divided by the number of instructions. However, this often leads to a large underestimation of the MLP, because misses are clustered, i.e., they occur in bursts. Furthermore, for some benchmarks, this naive approach overestimates MLP, because there are many dependences between loads (e.g., pointer-chasing code). Because the LLC miss component is in many cases one of the largest components, inaccurately estimating MLP often leads to inaccurate performance estimations.

An examination of the distribution of LLC misses showed that most of the burstiness is caused by cold misses, i.e., the first access to a certain cache line. Capacity and conflict misses are usually more spread out. So, for each unique data address, we record the position of the first load that accesses this address. When we apply the model we know the cache line size, and so we can locate the cold misses from the profile containing the first accesses to a data element. Alternatively, the location of the cold misses can also be calculated in the profiler, using multiple common cache line sizes. This reduces the generality of the profile, but it also reduces its size, because we don't have to keep all unique addresses, only the unique cache lines.

Using this profile, we calculate the distribution of the number of cold misses in a window equal to the ROB size. For the rest of the misses (as calculated by StatStack), we assume a uniform distribution. Furthermore, we take into account dependences by calculating the average number of dependence paths in the ROB as the ROB size divided by the average dependence path length (which is calculated by our profiler). We then use a probabilistic model to calculate the probabilities that loads belong to the same dependence path, i.e., the probability that they depend on each other. This further reduces the MLP. This technique leads to fairly accurate MLP estimations, as we will show in Section IV.

We found that for some applications, due to their bursty memory behavior, the memory bandwidth is often not sufficient, resulting in bus overloading and queueing up of memory operations. To model this, we assume that the number of concurrent misses is equal to the MLP on average, and that those misses access the memory at the same point in time.

Therefore, the first miss has a bus latency equal to the bus transfer time, i.e., the size of a cache block divided by the bandwidth. The second concurrent miss has to wait until the first miss frees the bus, so its bus latency equals twice the bus transfer time. And the third miss has a bus latency of three times the bus transfer time, etc. The average bus latency for MLP concurrent accesses is therefore

$$c_{bus}(\text{MLP}) = \frac{\text{MLP}+1}{2} c_{transfer} \quad (5)$$

We use linear interpolation to deal with non-integer MLP numbers.

F. Power modeling

Designing application-specific processors to improve energy-efficiency requires a power model. To estimate power, we use the McPAT tool [14], which requires the configuration of the processor and the activity factors (i.e., the number of accesses) for each component. The result is an estimation of the power and energy consumption.

For our model, we deduce the activity factors from the analytical performance model, instead of measuring them in simulation. Many of the inputs are already measured by the profiler for the performance model, such as the number of (micro)instructions and the instruction type mix. Other inputs are generated by the performance model, such as the number of misses in each level of the cache and in the branch predictor. However, some inputs are not required for the performance model, because it is assumed that they have no impact on performance, but are needed for the power model, because they cause more accesses. The most important examples are store misses and writebacks. Store misses can be generated by StatStack, as explained before. The number of writebacks is more difficult to estimate, but we noticed that they have a very small impact on total power consumption, so we choose not to model them.

IV. Accuracy and Speed Evaluation

A. Experimental setup

We implemented our profiler in Pin [24]. The profiler records the following statistics:

- Micro-operation count and instruction type mix, to calculate the average instruction latency.
- Average and critical dependency path for multiple ROB sizes (up to 1024), to calculate the effective dispatch rate, the branch resolution time and model dependences for the MLP estimation.
- The position of the first accesses to each unique data address, for the MLP estimation.
- The distribution of the number of loads per window, for multiple window sizes (up to 1024), also for the MLP estimation.
- Branch entropy for different local and global history sizes, for the branch miss component.
- Reuse distance profile for data loads and stores, and for instruction addresses, to estimate the number of cache misses at all cache levels.

TABLE I. Core configuration design space. Default values for the low-end, middle, and high-end cores are indicated in bold.

Parameter	Low-end	Middle	High-end
Dispatch width	**2**	**4**	**6**
ROB entries	32 - 48 - **64**	96 - **128** - 160	128 - 192 - **256**
Res. stations	1/3 - **1/2** - 2/3 of the ROB size		
Branch predictor	**pentium-M predictor [28]** - gshare - global predictor		
IL1 cache	**32 KB, 4-way, latency 1 cycle**		
DL1 cache	**32 KB, 8-way, latency 4 cycles**		
L2 cache	128 KB - **256 KB** - 512 KB **8-way, latency 8 cycles**		
L3 cache	1 - 2 - **4** MB	4 - 6 - **8** MB	8 - 12 - **16** MB
	16-way, latency 30 cycles		
Memory BW	**8 GB/s**		
Memory latency	**120 cycles**		

Because some of the profiles take some time to compute (e.g., the critical dependency path and reuse distance profile), we use sampling to reduce the profiling time. For example, for the critical dependency path, we sample every one out of 1,000 sequences of 1,024 instructions. This sampling technique introduces on average 5% error on the length of the critical path and less than 0.1% error on the instruction mix. We checked that it does not have a noticeable impact on the accuracy of the model. As a result, the profiler tool can profile an application at 2 MIPS (million instructions per second) on average, which is about $10\times$ faster than the average single-core simulation speed of Sniper.

For the reference detailed simulations we use the new cycle-level instruction-window centric core model in Sniper, which has been validated against real hardware [25]. Power measurements are done using the McPAT script included with Sniper for a 45nm technology.

We evaluate the model on the 29 SPEC CPU 2006 benchmarks with reference inputs. Because we have to simulate all of the designs to determine the accuracy of the model, we created a 1 billion instruction simpoint [26] for each benchmark. We use the Pinball technology [27] to create checkpoints at the beginning of the simpoint.

To show that the model is accurate across a large design space, we defined a broad set of processor configurations, see Table I. We split this set into three categories, low-end (dispatch width of 2), middle (dispatch width of 4), and high-end (dispatch width of 6) cores. The range of the sizes of the different components are in the table, the default value for each category is shown in bold. The parameters that are not mentioned in the table are equal to those of the standard Nehalem core configuration as included in the Sniper distribution. There are 729 different configurations in this design space.

To illustrate the speedup obtained by using an architecture independent profile, we calculated how much time it takes to evaluate the full design space for all benchmarks. For simulation, this takes 1,223 days of (serial) computation at 200 KIPS. For the original micro-architecture dependent profiler, where we have to simulate the 3 branch predictor configurations and 21 different cache hierarchies (3 settings for the L2 cache and 7 for the L3 cache) for all benchmarks, this takes 4.2 compute days, assuming that cache and predictor simulation executes at 2 MIPS. For the micro-architecture independent profiler, the time is shortened to only 4 *hours*, a $25\times$ speedup versus the micro-architecture dependent profiler.

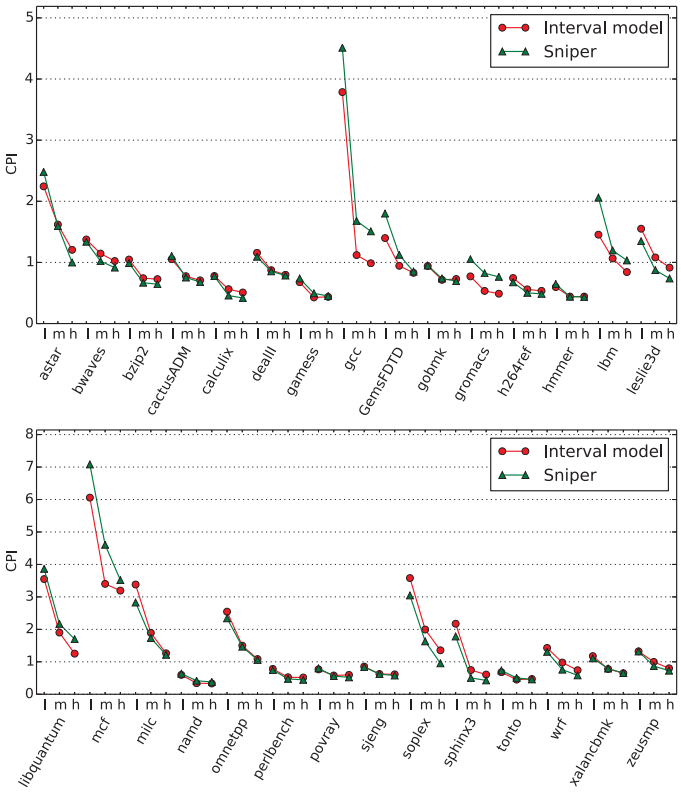

Fig. 3. Micro-architecture independent interval model CPI estimation versus simulated CPI using the Sniper detailed simulator, for three core configurations (l=low-end, m=middle, h=high-end).

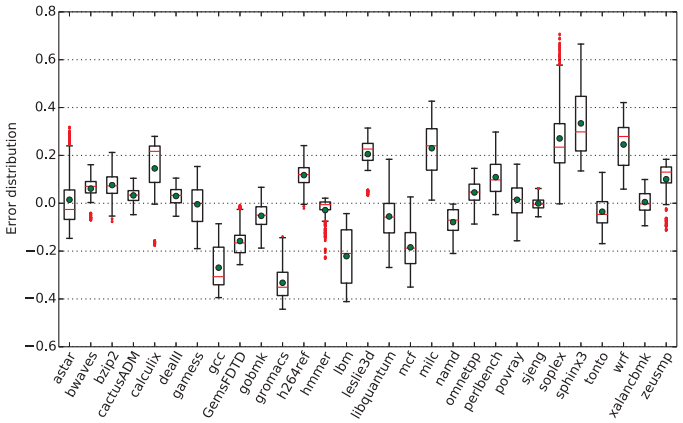

Fig. 4. Performance estimation error distribution per benchmark.

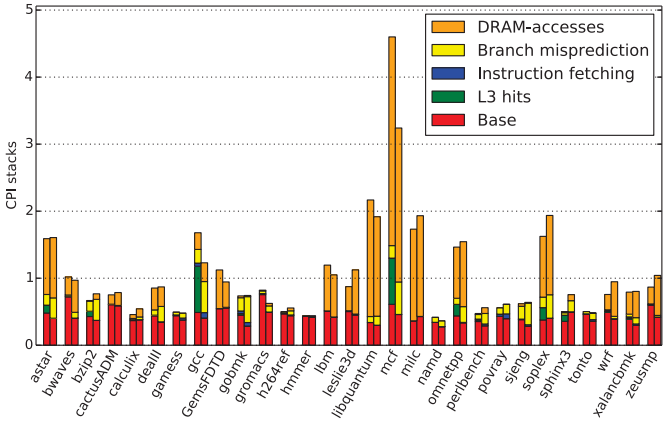

Fig. 5. CPI stacks for the middle configuration, as obtained by the Sniper detailed simulator (left) and as calculated by the model (right).

B. Accuracy

Figure 3 shows the CPI per benchmark for the default low-end, middle and high-end configuration, as estimated by the micro-architecture independent interval model and as simulated by Sniper. For many benchmarks, the model estimations are very close to the simulated values. For other benchmarks (e.g., gcc, lbm, mcf, and sphinx3), the estimations are a bit off, but the performance trend tracks well.

Figure 4 shows the distribution of the performance estimation error across the full design space. There is one box-and-whiskers plot per benchmark: the box is the range between the first and third quartile, the horizontal line in the box is the median, the dot is the mean, the whiskers cover all points within 1.5 inter-quartile distances outside the box, and the points represent the outliers. Although some benchmarks have high maximum errors (up to 70% for soplex), most of the benchmarks have an average error under 20%. Furthermore, for the benchmarks with large positive or negative errors, the error is consistently positive or negative, meaning that there is a component that is always over- or underestimated. A consistent error can be not harmful when comparing configurations: if both configurations have the same error, their relative difference is still correctly estimated. The average of the absolute value of the error over all benchmarks and all configurations is 13%, which is only 2% higher than the model with micro-architecture dependent inputs. This means that the error is mainly attributed to the assumptions of the model, rather than to the micro-architecture independent inputs.

In order to further investigate the sources of error, Figure 5 shows the CPI stacks as obtained by Sniper (left) and as calculated by our model (by showing each term of Equation 1 separately) for the default middle configuration. The base component is the performance in absence of misses, the branch misprediction component is the penalty due to branch mispredictions, the instruction fetching component is the performance loss due to instruction cache misses, and the DRAM-accesses component is the time spent in DRAM due to L3 cache misses. The Sniper CPI stacks have an extra L3 hit component that is not part of the model. This component is the penalty caused by loads that hit in the L3 cache (after missing in the other levels), and cause the ROB to fill up and stall the processor. The interval model assumes that all misses in the intermediate levels are hidden by out-of-order execution, and that only accesses to main memory cause a penalty. In the evaluated configuration, the L3 cache has a hit latency of 30 cycles, meaning that an ROB of 128 entries is almost filled if instructions are dispatched at 4 per cycle during these 30 cycles. As a result, L3 hits can sometimes cause a penalty, especially if they are chained through dependences. However, we found that most of the L3 hit penalty is still hidden, and adding an extra L3 hit term to the interval model causes significant overestimations of the execution time.

In general, most of the components are estimated quite

accurately, meaning that our overall accuracy results are not embellished by overestimating one component and underestimating another. The largest error occurs for benchmarks where the L3 hit component is large (gcc and mcf), but for all other benchmarks, this component is nonexistent or very small. Adding a crude L3 hit component to the model partly reduces the errors of gcc and mcf, but causes larger errors for the other benchmarks. The component with the largest error is the DRAM access component. This component is the last term in Equation 1, and contains 3 estimated parameters: the number of misses, the MLP and the bus queueing time. The error on this component is therefore a result of modeling errors in each of these 3 parameters, exacerbated by the high latency of memory accesses. In particular, the MLP is hard to model, due to subtle effects related to dependences and the exact position of the misses, but our relatively simple approach tracks the differences in MLP between the benchmarks quite well. For example, modeling no MLP for mcf leads to an error of 125%, while our model estimates an MLP of 3.34 (compared to a real MLP of 2.46), reducing the error to 27%.

Figure 6 shows the power estimations for the three configurations, compared to the Sniper+McPAT results. Although the 3 configurations are fixed, power consumption varies significantly over the different benchmarks due to different activity factors. For example, the power of the high-end core ranges between 15 W and 35 W. The model tracks this trend fairly accurately. Figure 7 shows the power estimation error distribution per benchmark. We see similar trends as for the performance estimations: the average error is for many benchmarks within 10%, and the benchmarks with a larger error have either a consistently positive or negative error. The average absolute error is 7%.

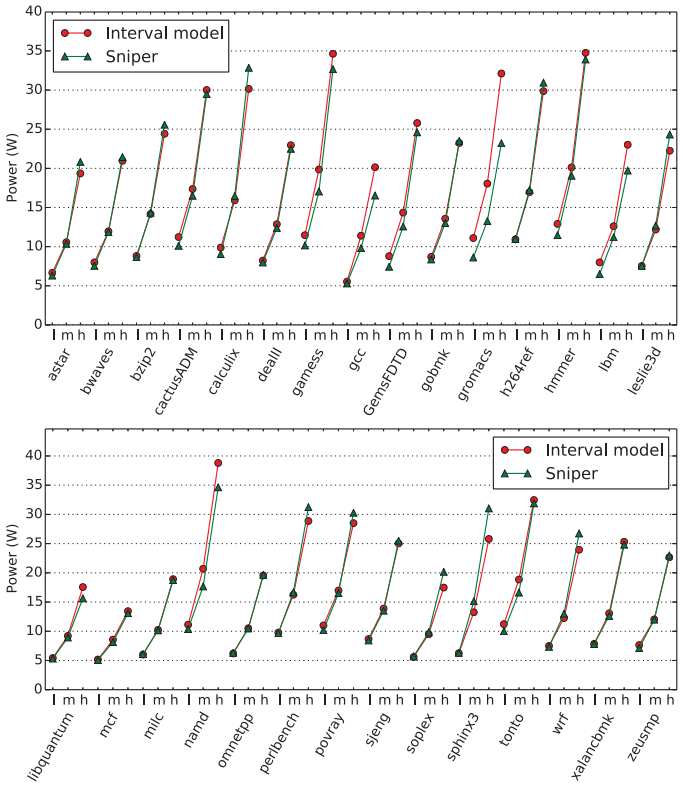

Fig. 6. Micro-architecture independent interval model power estimation versus simulated power using the Sniper detailed simulator, for three core configurations (l=low-end, m=middle, h=high-end).

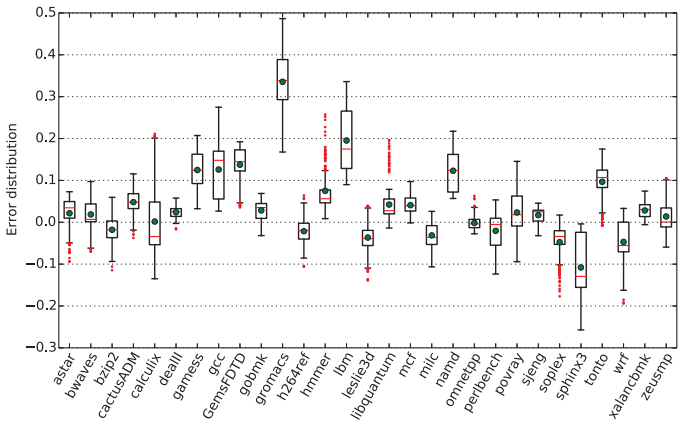

Fig. 7. Power estimation error distribution per benchmark.

V. Design Space Exploration Examples

A. Designing cores within a power budget

Our model can be used to efficiently explore design spaces and determine the most optimal processor configuration within given constraints. To prove this, we set up an example design use case: we try to find the best performing configuration within various power budgets.

We first show that optimizing core configuration for individual applications indeed leads to better overall performance compared to selecting a single design that performs best on average over all applications within the same power budget. Figure 8 shows the average CPI (lower is better) across all applications for the single optimal design (left) and for the application-specific designs (right). It is clear that selecting application-specific cores leads to a higher performance (lower CPI) than selecting a single design, especially for the low power budgets. For example, for the 15 W power budget, the single best design is a low-end design, to avoid a power overshoot for one particular benchmark, while the 28 other benchmarks can benefit from the higher performance of a middle and even high-end core, and still remain within 15 W.

The previous results were generated using the simulated results to show the potential of application-specific core design. Now, we use the model to find the optimal designs without simulating the full design space. Because there is some error on the power estimations, we use the model to select five possibly optimal designs out of the design space of 729 configurations, instead of a single configuration. We do this by also selecting the best performing designs for power budgets that are 20% and 10% under the targeted power budget, as well as power budgets that are 10% and 20% higher, next to the one with the exact power budget. This method ensures that if the power consumption is somewhat over- or underestimated by the model, we can still select configurations that are under, but close to the power budget. We then simulate these five designs using cycle-level simulation, and select the one that meets the power constraints and has the best performance. Although this technique still requires some detailed simulations, the number of simulations is drastically lower compared to exhaustively

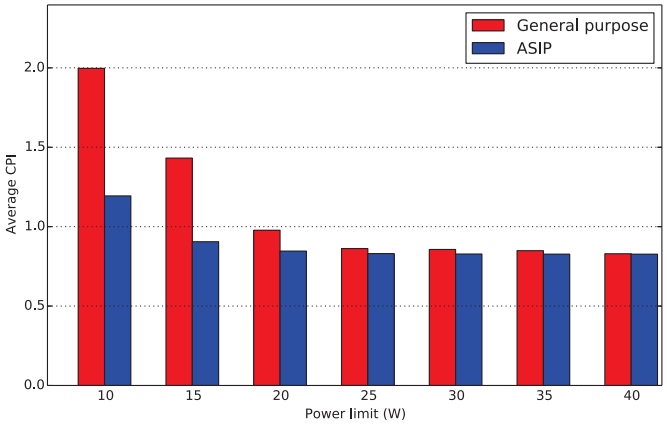

Fig. 8. Average CPI (lower is better) for the single best configuration (general purpose) versus the best configuration per application (ASIP) for different power budgets.

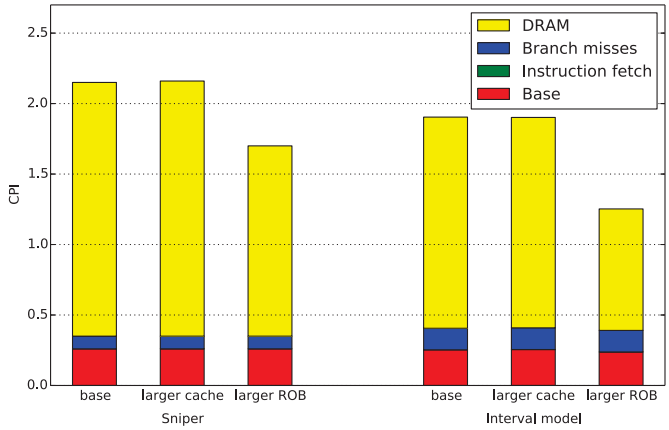

Fig. 9. CPI stacks measured by Sniper and estimated by our model for libquantum. 'Base' is a high-end configuration with 128 ROB entries and a 8 MB L3 cache. 'Larger cache' is the same configuration, but with 16 MB L3 cache, and 'larger ROB' has an 8 MB cache, but a ROB of 256 entries.

TABLE II. NUMBER OF BENCHMARKS FOR WHICH THE DESIGN SPACE EXPLORATION TECHNIQUE RESULTS IN A THE OPTIMAL DESIGN, OR A SUBOPTIMAL DESIGN WITHIN 1%, 5%, 10% OR MORE THAN 10%.

Power budget	Optimal	$< 1\%$	$< 5\%$	$< 10\%$	$> 10\%$
10 W	9	6	5	3	11%,13%
15 W	6	6	12	1	11%,12%,18%,36%
20 W	10	7	10	1	12%
25 W	10	7	10	2	–
30 W	11	7	9	2	–

simulating the full design space.

Using this technique, we were able to find configurations that are all within the imposed power budget. Table II shows for each power budget the number of benchmarks for which we find the exact same configuration as simulating the full design space in detail. Then we show the number of benchmarks where the resulting configuration performs less than 1% worse than the actual optimal configuration, followed by less than 5% and less than 10%, and in the last column, we show how much the remaining benchmarks are off (i.e., the ones that have more than 10% lower performance). We only show upto a power budget of 30 W, because the previous graph showed that there is little to gain for the larger power budgets. Note that for 4 benchmarks, there is no configuration that consumes less than 10 W, explaining the smaller number of benchmarks. This was also correctly detected by our model.

For the vast majority of the benchmarks, we find a configuration that is within 5% of the optimal configuration. The explanation for the outliers at the small power budgets is that there are a lot of configurations that are just under and above this budget. Therefore, by only picking 5 points, we sometimes miss the configurations that are just beneath the power budget and have the highest performance. We checked that picking more points for the 10 W and 15 W power budget leads to finding better configurations.

B. Optimizing performance

In this section, we show that using a model can provide more insight in how performance can be increased. We consider the case of libquantum. The left CPI stack in Figure 9 is measured using a Sniper simulation on a high-end configuration with 128 ROB entries and a 8 MB L3 cache (base). Because there is a large DRAM component, it is intuitive to increase the cache size for better performance. However, our model (right part of the graph) shows that there is little to gain in increasing the cache to 16 MB, because the number of misses does not significantly decrease. On the other hand, increasing the ROB size to 256 entries leads to a larger MLP, effectively decreasing the DRAM component and improving performance. This trend is confirmed by simulating the larger cache and the larger ROB using Sniper. However, to reach this conclusion using simulation only, we need at least 3 simulations, and probably more as it is not immediately clear that increasing the ROB size leads to the largest performance improvement (e.g., an architect might first try different cache sizes, or increasing the memory bandwidth). Using our model, we can obtain the same conclusion with one profiling step. To be sure, the hints made by the model can be confirmed by detailed simulation, but this only requires a few simulations compared to exploring all possibilities using simulation.

VI. CONCLUSIONS AND FUTURE WORK

Application-specific processor cores can substantially increase energy-efficiency. Designing application-specific processors requires a fast design space exploration tool, as the design process needs to be redone for every application. We demonstrated an analytical performance and power model, based on micro-architecture independent application profiles to enable the evaluation of large design spaces using a single application-specific but architecture-independent profile. Performance and power are estimated with an average error of 13% and 7%, respectively, compared to detailed simulation. We show that the model can be used to find the best performing core configuration for each application within a fixed power budget, resulting in a higher average performance compared to selecting one single design that has the best average performance across all applications within the same power budget.

Our model currently only provides performance and power estimations for a single-threaded program on a single core, which can be a starting point for determining (an) optimal core configuration(s) for designing a multicore processor. However, we plan to extend the model to estimate the performance and

power of a full multicore processor, including shared caches and shared memory components, and of multi-threaded (SMT) cores. We also plan to model the impact of synchronization and cache coherence on the performance and power consumption of multi-threaded applications. We can also extend our work with accelerator models [17].

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive and insightful feedback. Sam Van den Steen is supported through a doctoral fellowship by the Agency for Innovation by Science and Technology (IWT). Stijn Eyerman is supported through a postdoctoral fellowship by the Research Foundation – Flanders (FWO). Additional support is provided by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295, as well as by the European Commission under the Seventh Framework Programme, Grant Agreement no. 610490.

REFERENCES

[1] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[2] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 205–218.

[3] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376.

[4] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A Mechanistic Performance Model for Superscalar Out-of-Order Processors," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 2, pp. 42–53, 2009.

[5] B. Lee and D. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 185–194.

[6] E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana, "Efficiently exploring architectural design spaces via predictive modeling," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 195–206.

[7] P. Emma, "Understanding some simple processor-performance limits," *IBM Journal of Research and Development*, vol. 41, no. 3, pp. 215–232, 1997.

[8] T. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004, pp. 338–349.

[9] X. E. Chen and T. M. Aamodt, "Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2008, pp. 59–70.

[10] A. Hartstein and T. R. Puzak, "The optimal pipeline depth for a microprocessor," in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002, pp. 7–13.

[11] S. Eyerman, K. Hoste, and L. Eeckhout, "Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011, pp. 216–226.

[12] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2010, pp. 307–318.

[13] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000, pp. 83–94.

[14] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480.

[15] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*, 2003, pp. 93–104.

[16] Y. S. Shao and D. Brooks, "ISA-independent workload characterization and its implications for specialized architectures," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 245–255.

[17] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*, 2014, pp. 97–108.

[18] T. Karkhanis and J. E. Smith, "Automated design of application specific superscalar processors: An analytical approach," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007, pp. 402–411.

[19] S. Eyerman, J. E. Smith, and L. Eeckhout, "Characterizing the branch misprediction penalty," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2006, pp. 48–58.

[20] S. De Pestel, S. Eyerman, and L. Eeckhout, "Micro-architecure independent branch prediction modeling," in *Proceedings of the International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2015.

[21] D. Eklov and E. Hagersten, "Statstack: Efficient modeling of lru caches," in *Proceedings of the International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010, pp. 55–65.

[22] E. Berg and E. Hagersten, "Fast data-locality profiling of native execution," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2005, pp. 169–180.

[23] A. Sembrant, D. Black-Schaffer, and E. Hagersten, "Phase guided profiling for fast cache modeling," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO)*, 2012, pp. 175–185.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, 2005, pp. 190–200.

[25] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 28:1–28:25, 2014.

[26] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002, pp. 45–57.

[27] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th annual international symposium on Code Generation and Optimization (CGO)*, 2010, pp. 2–11.

[28] V. Uzelac and A. Milenkovic, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 207–217.