Wire-driven Microarchitectural Design Space Exploration

Mongkol Ekpanyapong, Sung Kyu Lim, Chinnakrishnan Ballapuram, and Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332, U.S.A.
{pop,limsk,chinnak,leehs}@ece.gatech.edu

***Abstract*— In this paper, we propose an interconnect-driven framework that performs an efficient and effective design space exploration for deep submicron processor architecture design. At the heart of our framework named AMPLE are wire delay-driven microarchitectural floorplanning and adaptive parameter tuning schemes that address interconnect issues with high exploration efficiency and accuracy. Our framework significantly outperforms the commonly used brute-force and Simulated Annealing methods in terms of exploration time efficiency as well as the performance and area quality for a large design space.**

I. Introduction

For today's high performance processor design, it is crucial to fully understand the implications between emerging wire delay issues and each microarchitectural component for a given type of applications. For example, for applications that perform intensive floating-point operations, the floating-point units should be placed closer to the execution units and ALUs. On the other hand, memory-bound applications should have their datapath closer to caches. To address the design complexity effectiveness for deep submicron processor, we propose a new methodology for design space exploration called *Adaptive Microarchitectural Planning Engine* (AMPLE), which identifies the most complexity-effective processor configuration, in terms of performance and performance/area improvement, in a highly efficient manner for a given suite of target applications.

AMPLE performs interconnect-driven microarchitectural floorplanning [1] to reduce the communication latency by placing high bandwith communication modules adjacent to each other. We then use this new floorplan-based latency information to tune various architectural parameters to further optimize the design objectives. AMPLE can effectively alleviate wire delay problem during design space exploration. By increasing the sizes of modules, we may improve the overall performance from the incremented functionality. However, there is a point of diminishing return due to the increase of wire delay for accessing these modules. In order to precisely determine this break-even point, AMPLE fine-tunes the parameters based on accurate communication latency given by floorplanner. In addition, the quality of the proposed parameter set is accurately measured from a cycle accurate simulator and is used for the next set of parameters.

II. AMPLE Exploration Algorithm

A. Overview of the Approach

To explore all possible permutation of different microarchitecture design parameters is extremely tedious and time-consuming, if not impossible, requiring enormous computing cycles and patience. Some recent research efforts were attempted and focused on improving the efficiency of the search process with various heuristics [2], [3]. In our AMPLE framework as illustrated in Figure 1, the system first reads the technology parameters, an initial machine description, and the target application benchmark program as input. For cache-like or buffer-like structures, the areas and module delays are estimated using analytical tools, in our case, CACTI [4] from the HP Western Research Labs. For non-memory components, GENESYS [5] is used. At the same time, we profile the target application to collect the statistical interconnection traffic using Simplescalar [6], a cycle-accurate architecture simulator.

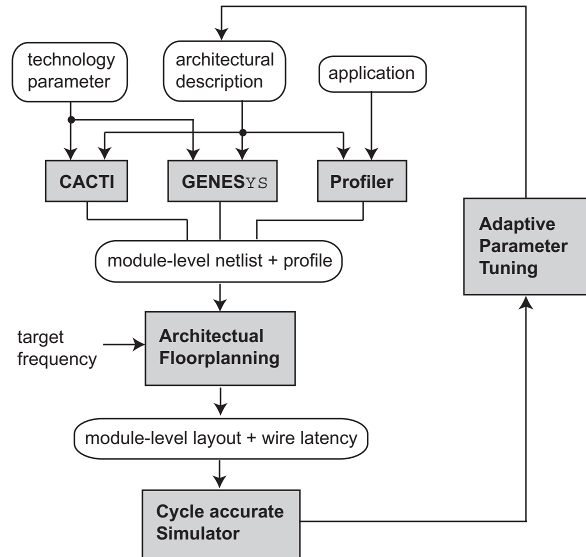

Fig. 1. AMPLE microarchitecture design space exploration framework

All the module information, statistical interconnection traffic, and frequency target range are then fed into the profile-guided floorplanner with an objective of minimizing the latency (i.e., communication distance) of the most frequent communication links between modules. We use our profile-guided floorplanning algorithm developed in [1] for this purpose. Afterward, the inter-modular latencies are derived based on the newly generated floorplan. The cycle-accurate simulator is again used to evaluate the architecture performance using the new latency cycles. Given the cycle time, the actual execution time can be computed. Along with the machine description parameters, the floorplan and new latency values are given to the AMPLE search engine. Our adaptive parameter tuning engine, based on the gradient search heuristic, calculates and determines a new set of

0-7803-8834-8/05/$20.00 ©2005 IEEE.

architecture parameters and iterates through the entire process until a satisfactory parameter suite is found.

B. Microarchitecture Model

Ten microarchitecture parameters were investigated for our target microarchitecture design: *Width* is the issue width (range 1-8). *BTB* is the branch target buffer that stores target branch addresses for predicted taken branches (range 128-1024). *RUU*, the Reservation Update Unit [7], combines the functionality of the reservation station and the re-order buffer for out-of-order execution (range 32-1024). *LSQ* is the load/store buffer for resolving address conflicts between loads and stores (range 16-512). A tri-level cache includes a split L1 I-cache and L1 D-cache (both range 1KB-64KB), a unified L2 (range 8KB-1MB) and a unified L3 (range 64KB-1MB). *ALU* (range 1-8) and *FPU* (range 1-8) are the arithmetic logic unit and the floating-point unit. The potential values for these parameters are commonly seen in the state-of-the-art high performance microprocessors. The range of inter-module access latency values between major modules in clock cycles are: branch predictor penalty (5-11), pipeline depth (15-22), inter ALU communication (1-9), inter-FPU communication (6-28), RUU access from ALU (1-9), RUU access from FPU (2-29), IL1 access (2-12), DL1 access (2-11), IL2 access (6-44), DL2 access (6-39), L3 access (26-128).

C. Gradient Search

Our adaptive parameter tuning algorithm is presented in Figure 2. First, the gradient search algorithm was initialized with Smart_Start() algorithm which, based on the characteristics of each application, assign the initial microarchitecture parameter values for the starting search point. A good starting point could reduce the overall design space exploration time substantially as the gradient search method tends to move toward a local optimal point in the search space. If the starting search point is given at a location somewhere close to the optimal point, then a smaller number of steps will be needed. Given these initial conditions, we perform Microarch_Planning(), which consists of technology parameter extraction, microarchitectural profiling, floorplanning, and cycle-accurate simulation, as depicted in Figure 1. Before entering the iterative gradient search for all N microarchitectural components, the relative importance of each microarchitectural parameter p for the target application is determined using Priority_Search() algorithm. In essence, the most sensitive microarchitectural parameter has the highest priority to be tuned in the main search loop.

The gradient search then enters the iterative design space exploration loop based on the priority determined by Priority_Search(). For each iteration i, the last known best microarchitecture parameter is used as the starting point. The value of the microarchitectural parameter p at iteration i, denoted $para[p, i]$, is determined based on two factors: *initial stepping factor* ($= \alpha$) and *stepping size* ($= \mu$) [8]. Note that prior to the microarchitectural planning, a parameter pruning algorithm Para_Pruning() is always called to rule out invalid parameters. The initial stepping factor α value is used only in the first iteration to provide the initial gradient.

Adaptive Parameter Tuning Algorithm
Smart_Start();
Microarch_Planning();
PS = Priority_Search();
for (parameter $p = 1$ to N in PS)
    $para[p, 1]$ = last known best parameter;
    $para[p, 2] = (1 + \alpha) \times para[p, 1]$;
    Ceiling$_p$($para[p, 2]$);
    Para_Pruning($para[p, 2]$);
    Microarch_Planning();
    Calc_GFR();
    Calc_Gain();
    $i = 3$;
    while ($gain[p, i-1] > threshold$ or not cyclic)
        compute $para[p, i]$ using Equation 2;
        if ($gain[p, i-1] > 0$)
            Ceiling$_p$(para$[p, i]$);
        else
            Floor$_p$(para$[p, i]$);
        Para_Pruning($para[p, i]$);
        Microarch_Planning();
        Calc_Gain();
        i++;

Fig. 2. AMPLE design space exploration algorithm

Once inside the while loop, i.e., the main gradient search loop, a stepping size μ is then used to increase or decrease the microarchitecture parameter (compute $para[p, i]$, see Equation (2)) depending on whether a constructive gain or a destructive gain is obtained from the last iteration. The Calc_Gain() function in Figure 2 calculates the gain using the following equation:

$$gain[p, i] = \frac{exec_time[p, i-1]}{exec_time[p, i]} - 1 \quad (1)$$

where $exec_time[p, i]$ denotes the overall performance of the microarchitecture with the current parameter p set to $para[p, i]$ at iteration i. Essentially, the gain represents the overall performance improvement (or degradation) in percentage. If the gain is negative, then the search is apparently heading toward the wrong direction and needs to be reversed. Otherwise, it should continue by increasing the $para[p, i]$ for the next iteration based on the following equation:

$$\begin{aligned} para[p, i] &= para[p, i-1] + \mu \times gain[p, i-1] \\ &\times (para[p, i-1] - para[p, i-2]) \end{aligned} \quad (2)$$

The amount of increase is the product of the stepping size μ, the gain and the parameter deviation from the previous two iterations [8].

Note that the stepping size μ determines the search quality and/or search time. A finer stepping size will lead to a better quality solution at the cost of a longer search time. On the other hand, if the stepping size is too big, the result may diverge and/or miss the optimal opportunity. Whenever a new $para[p, i]$ is generated, the Ceiling or the Floor function of the corresponding microarchitecture component p is used to clamp the parameters to meet the particular criteria of a microarchitecture component. For example, the number of the cache sets must be in the power of two for indexability, or the number of the ALU or FPU has to be an integer.

The whole iterative process for parameter p continues

until one of the following two conditions is met: (1) the overall performance gain drops below a given threshold, or (2) the exploration process is going cyclic due to hitting a repetitive microarchitecture parameter value of p that was explored earlier. In addition, in the first iteration, we also calculate a ratio called *Gradient First-order Ratio* (GFR) using Calc_GFR(), which gives us an estimation of how relevant a microarchitecture component p is with respect to the target applications.

D. Smart Start Algorithm

Applications are generally classified into three categories [9] — processor-bound applications, cache-sensitive applications, and bandwidth-bound applications. We apply the same classification scheme in the Smart_Start() algorithm to determine the initial conditions for each benchmark program. The initial microarchitecture parameter values are predetermined based on the class it belongs to. For example, for the processor-bound applications, a wider issue width, more ALUs, and a larger instruction cache and BTB are used. Using the similar logistics, the cache-sensitive applications will be allocated with a larger L1 and L2 data cache. Finally, for the bandwidth-bound's class, it is hard to adjust the module parameters on this class since on-chip microarchitecture modules do not really improve off-chip memory bandwidth. Toward this class, we enlarge the L3 size and load/store queue as the initial conditions. The microarchitecture parameters picked by Smart_start() for the three different application classes are listed in Table I.

E. Priority Search

The priority among all microarchitectural parameters is also based on the benchmark classification. Since "high impact parameters" are more sensitive to the performance, these high impact parameters are tuned first. For the processor-bound class, issue width, instruction cache size, BTB size, and the number of ALUs are given higher priority. For the cache-sensitive applications, L1 data cache, L2 cache, and load/store queue have higher priority. For the bandwidth-bound class, all three level cache and load/store queue are more focused. Note that if the search time is constrained, the microarchitecture parameters with lower priority will not be explored at all for less performance impact is to be expected.

If the search time is limited, spending more time on higher impact parameters is obviously more rewarding. The best way to identify the critical parameters is to explore the entire search space and compute the correlation metric. Nevertheless, with a large search space, this method is technically impractical. A more efficient alternative is to use some sample set from all possible search space and then calculate the correlation based on that sample set. This method is more practical, but still require a non-trivial search time. Especially, acquiring a good sample set that is the most representative for the entire search space is not trivial. Hence, running more search points will bring us closer to such a good sample set. Once the correlation metric is identified, we can use it to perform an extensive search on the critical microarchitecture parameters in lieu of searching for the entire search space. This might not result in the true optimal solution, but can reduce the search time significantly. Here we propose a Gradient First-order Ratio (GFR) metric computed as follows:

$$GFR(p) = \frac{gain[p,2]}{\max_{q \in N} gain[q,2]} \quad (3)$$

Although it is not as good as the correlation function, it can be used to roughly estimate how crucial each parameter is. The GFR metric requires running the search only twice the number of parameters (using first two runs). The GFR of a parameter p is calculated as the ratio of the gain of the first iteration of parameter i and the maximum among all the parameter gain.

F. Search Pruning

During the design space exploration, many unrealistic combinations of architecture parameters can be omitted. For example, in order to maintain cache inclusion property [10], the L1 cache size should be less than that of the L2. The following summarize three guidelines we follow in order to prune the search space: cache size L1 $<$ L2 $<$ L3, issue width $\geq$ the number of ALU, no search in floating-point units for integer applications. Note that this search pruning approach is also applied to the brute-force search that we use as a reference baseline. Another criteria in AMPLE is to enforce the lower and upper bound constraints. First of all, for a given processor design, only limited number of resources is available. An upper bound constraint guarantees that a particular microarchitecture module will not overgrow to become an overkill. For example, designing a 20MB or larger L3 cache on-die might not be very convincing for a $90nm$ processor. On the other hand, a lower bound constraint is imposed so that the essential microarchitecture module will not be completely removed. For example, at least one ALU is needed.

III. Experimental Results

We assume a 10 GHz processor designed with a $50nm$ feature size as projected by ITRS. The technology parameters based on $50nm$ are used in our technology scaling models. We applied search pruning to our brute-force method in order to reduce the number of simulations. The simulated annealing approach is based on the algorithm proposed in [2]. For our AMPLE engine, the following constants are used: initial stepping factor $\alpha = 0.10$, $gain\ threshold = 100$, and the stepping size $\mu = 1$. We show the design space exploration results of four different methods including *best*, *sa*, *gra*, and *grad II*. All results were reported in absolute performance (i.e., $IPC \times clock_period$) and were normalized to the average of the *brute-force* search method. *best* represents the best processor design parameters reported by the brute-force search. *sa* uses simulated annealing approach. *gra* is based on our AMPLE search engine using performance as the objective function. *gra II* is using AMPLE technique with both the performance and die area as the design objectives. The reason that we used the average of the brute-force search as the baseline is because this is likely to represent the case of a random selection configuration.

The total search times are compared in Table II for three schemes–brute force, simulated annealing, and AMPLE. Runtime is reported in *hours*. Most part of the runtime is spent in cycle accurate simulation, where each simulation took about an

TABLE I

INITIAL CONDITION USED BY SMART_START()

Class	width	BTB	RUU	LSQ	L1 Icache	L1 D$	L2 U$	L3 U$	ALU	FPU
Processor bound	8	512	256	128	16K	8K	128K	0	6	4
Cache sensitive	4	256	128	256	8K	16K	512K	0	4	2
Bandwidth bound	4	256	128	128	8K	8K	128K	0	4	2

TABLE II

RUNNING TIME COMPARISON

bench	brute force		annealing		gradient	
	time	itr	time	itr	time	itr
164.gzip	314	384	34	43	29	36
175.vpr	406	384	50	43	42	35
181.mcf	209	384	29	43	18	36
254.gap	325	384	49	43	36	35
300.twolf	202	384	21	43	18	36
171.swim	830	768	62	43	45	33
179.art	1,561	768	126	43	86	38
189.lucas	396	768	22	43	15	32
Avg.	14.31		1.34		1.00	

hour. This translates into $2^{10} \times 1$ hour $\simeq 42$ days for the brute-force approach without any search pruning heuristics. The total runtime for brute-force scheme for all benchmark was 17 days on a 10-machine cluster. For brute-force search, 10 parameters are examined, and for each parameter, two different values are studied. From this, it can be seen that number of search during the brute force search is limited. On the other hand, simulated annealing and gradient search allows to examine each parameter with more different values. This can lead to a better solution especially if there are some parameters that have more performance impact than the others. In the case of simulated annealing, we chose an annealing schedule in such a way that the total runtime is similar to AMPLE. This allows us for a fair comparison of the solution quality.

Figure 3 shows that all four methods significantly outperform the *baseline*. *gra*, on the other hand, outperforms *best* by 12.7% (up to 2.12 times), *sa* by 14.1% (up to 1.45 times), and *gra II* by 32.4%. Note that finding the optimal solution is a hard problem since the latency among modules is based on location information from floorplanner. However, we believe that the set of optimal and nearly optimal parameters are highly localized. Gradient search allows us to discover these sets of points in the solution space faster. From Figure 4, we see that the *best* and *sa* requires about the same area, and it is about three times smaller than the *baseline*. The outcome of our *gra* algorithm requires one-sixth of the *baseline* area. In addition, the outcome of the *gra II* requires an order of magnitude less area than the *baseline*.

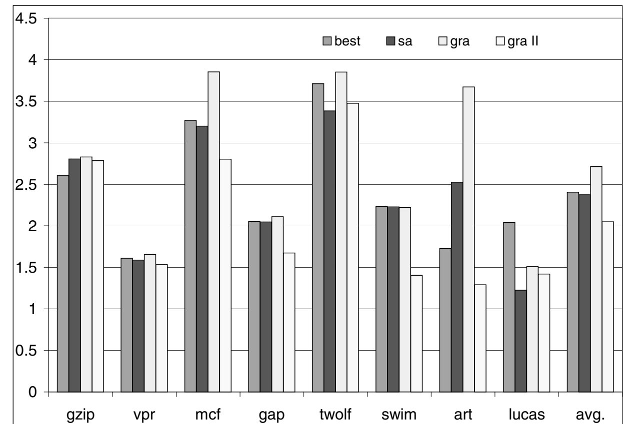

Fig. 3. Performance Speedup (baseline: *average brute force* = 1.0)

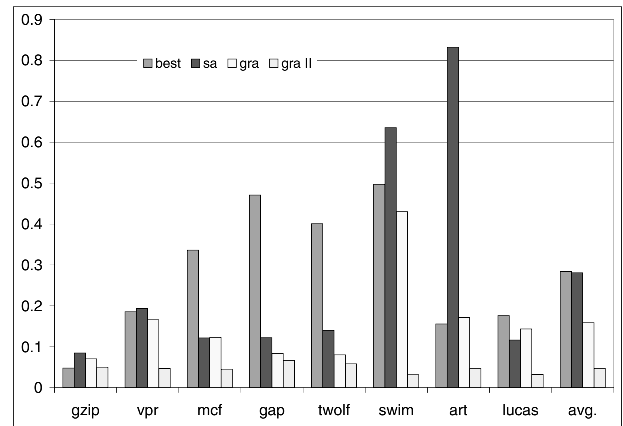

Fig. 4. Area comparison (baseline: *average brute force* = 1.0)

IV. CONCLUSIONS

We propose a framework AMPLE that enables an efficient design space exploration for deep submicron microprocessor design. Using AMPLE, we can identify a highly cost- and complexity-effective set of microarchitecture parameters within a reasonable amount of time.

REFERENCES

[1] M. Ekpanyapong, J. Minz, T. Watewai, H.-H. S. Lee, and S. K. Lim, "Profile-Guided Microarchitectural Floorplanning for Deep Submicron Processor Design," in *Proc. ACM Design Automation Conference*, 2004.

[2] J. Cong, A. Jagannathan, G. Reinman, and M. Romesis, "Microarchitecture Evaluation with Physical Planning," in *Proceedings of the 40th Conference on Design Automation*, 2003.

[3] T. Givargis, F. Vahid, and J. Henkel, "System-level Exploration for Pareto-optimal Configurations in Parameterized System-on-a-chip," *IEEE Transsactions on VLSI System*, vol. 10, no. 4, pp. 416–422, Decmeber 2002.

[4] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model," HP Western Research Labs, Tech. Rep. 2001.2, 2001.

[5] J. C. Eble, V. K. De, D. S. Wills, and J. D. Meindl, "A Generic System Simulator (GENESYS) for ASIC Technology and Architecture Beyond 2001," in *Int'l ASIC Conference*, 1996.

[6] T. M. Austin, "Simplescalar tool suite," http:/www.simplescalar.com.

[7] G. Sohi and S. Vajapeyam, "Instruction Issue Logic for High-Performance Interruptable Pipelined Processors," *Proceedings of the 14th Annual International Symposium on Computer Architecture*, 1987.

[8] B. Farhang-Boroujeny, *Adaptive Filters — Theory and Applications*. John Wiley and Sons, 1998.

[9] J. Huh, D. Burger, and S. W. Keckler, "Exploring the Design Space of Future CMPs," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[10] J.-L. Baer and W.-H. Wang, "On The Inclusion Properties for Multi-level Cache Hierarchies," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, pp. 73–80.