Magellan: A Search and Machine Learning-based Framework for Fast Multi-core Design Space Exploration and Optimization

Sukhun Kang and Rakesh Kumar

Coordinated Science Laboratory
1308 West Main St
Urbana, IL 61801

Abstract

In this paper, we treat multi-core processor design space exploration as an application-driven machine learning problem. We develop two machine learning-based techniques for efficiently exploring the processor design space. We observe that these techniques result in multi-core processors whose performance is comparable (within 1%) to a processor design that requires an exhaustive exploration of the design space. These techniques often take orders of magnitude *(a factor of 3800 at the minimum) less time for coming up with these processors. The benefits are up to 13% over intelligent search techniques that have been adapted to do multi-core design space exploration.*

We leverage the knowledge gained in this research to develop Magellan – a framework for accelerating multi-core design space exploration and optimization. Magellan can be used to find the highest throughput processors of a given type for a given area, power, or time budget. It can be used to aid even experienced processor designers that prefer to rely on intuition by allowing fast refinements to an input design.

1 Introduction

Conventional methodologies for designing a processor have relied heavily on large-scale simulations to evaluate the various architectural possibilities. However, simulations often take long and can limit the number of possibilities that can be considered for a given time budget.

This paper recognizes that an exhaustive simulation-based approach to explore the processor design space may not scale well for future heterogeneous multi-core processors. In this paper, we treat the processor design exploration problem as an application-driven machine learning problem (while also borrowing the idea from the uniprocessor space of using search techniques to do processor design) and develop techniques that are based on using application characteristics to eliminate processors that would be a bad match for the expected workload universe. We propose two machine learning-based algorithms to prune the search space. The proposed techniques can reduce the search space by up to 13% (9% on average) even over using search algorithms.

We leverage the knowledge gained in this research to develop Magellan – a framework for accelerating multi-core design space exploration and optimization. Magellan is presented with a library of cores that are used to create chip multiprocessors and a set of applications representative of the workload universe. Given this information, Magellan can be used to find the highest throughput processors of a given type for a given area, power, or time budget. It can be used to aid even experienced processor designers that prefer to rely on intuition by allowing fast refinements to an input design.

To put the work presented in this paper in context of previous work, there has been recent work on accelerating multi-core design space exploration either through statistical inference [5] or through predictive modeling [3]. Our approach is orthogonal to those approaches. Similarly, heuristic-based approaches have often been used to search through the design and optimization space of simple uniprocessors [2]. While several of those approaches can often be applied directly to the design of many-core processors, our work is different in that the machine learning-based techniques presented here are novel. Also, we present a new framework/tool for multi-core exploration and optimization.

2 Adapting Search Algorithms for Multi-core Design Space Exploration

Previous work [2] has modeled uniprocessor design as a search problem. One aspect of Magellan is implementing some of these search algorithms for multi-core design space exploration.

[0] A longer version of the paper appears as UILU-ENG-07-2212, a UIUC CRHC Technical Report (also indexed as CRHC-07-05)

978-3-9810801-3-1/DATE08 © 2008 EDAA

Steepest Ascent Hill Climbing

Exploring multi-core design using steepest ascent hill climbing involves evaluating all combinations of every neighboring core and using the best k-core combination for the next iteration. A neighboring core is defined as a core that differs in only one parameter and that too in the smallest granularity. For example, two cores that are identical in all parameters except in their icache sizes where the icache size of one is 8KB and the other is 16KB are neighboring cores.

Pseudo-Code

```
1:  S = initial k core configuration E = Evaluation of S MaxIPC = E
2:  BestConf = S
3:  while not stuck do
4:      N = neighbors of S
5:      for  i to maxnum ; maxnum = number of combinations of N do
6:          En = evaluations of N_i
7:          if  En > BestEn; if new N is better than best N so far then
8:              BestEn = En
9:              BestN = N_i
10:         end if
11:     end for
12:     if BestEn > MaxIPC then
13:         MaxIPC = Bestn
14:         BestConf = N_i
15:     end if
16:     S = N_i
17: end while
```

Genetic Algorithm

Our implementation of genetic algorithm for multi-core design space exploration involves 4 stages: Reproduce, Crossover, Mutation, and Natural Selection. Reproduce stage simply evolves the current processor to next neighbors. Crossover stage cross over the population and come up with new processors. Mutation stage randomly picks numbers of cores and mutate them to a random core. Finally, Natural Selection stage picks 4 top performing processors and use them in the next step of exploration.

Pseudo-Code

```
1:  S = initial k core configuration E = Evaluation of S
2:  for I = 1 : Kmax do
3:      RPool = Reproduce (S) ; find better neighbor of each k-core set
4:      CPool = CrossOver(ReproducePool) ; crossing over all the populations
5:      MPool = 4 Randomly mutated k-core of current population
6:      S = Top 4 set of k core configurations out of S, Rpool, Cpool,MPool
7:  end for
```

Ant Colony Optimization

We adapt ant-colony optimization technique to the multi-core processor design space exploration problem taking different paths every iteration in our search for the "optimal" processor (or the ones close to it).

Pseudo-Code

```
1:  S = initial k core configuration E = Evaluation of S MaxIPC = E
2:  BestConf = S Path = initial path
3:  while not stuck do
4:      for  i = 1 : Kmax do
5:          N = better neighbor of path[i]
6:          En = evaluations of N @
7:          if En > MaxIPC then
8:              MaxIPC=En
9:              BestConf = N
10:         end if
11:         if En != Eprevious then
12:             Path[i] = [N, En]
13:         end if
14:     end for
15: end while
```

The advantages and disadvantages of these search algorithms are fairly well-known. The reader is referred to a standard book on optimization techniques for a more through analysis of the differences between them.

3 Treating Processor Design as a Machine Learning Problem

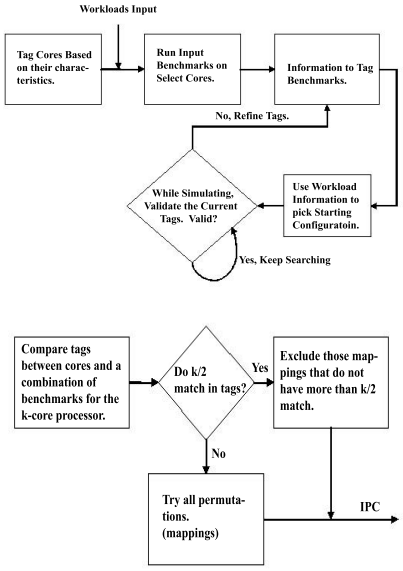

Figure 1. a) Flowchart of Machine Learning Technique And (b) the Optimization of Workloads to Cores Mapping

The proposed techniques for machine learning based-search for finding the best processor involves two phases, tagging cores and benchmarks while simulating, and searching. We start out by simulating the benchmarks on a small number of distinct cores that are picked according to their characteristics such that they are sufficient to tag all

the benchmarks. The performance and tag values are kept in arrays. As we begin searching through the solution space, we set the starting configuration according to the characteristics of the benchmarks that we have tagged already. Every iteration of search, we look at the combinations of cores that fit the benchmarks' characteristics. As the search proceeds, we run into cores that have not been seen yet. The performance of workloads on these combinations can potentially be used to further refine the benchmark tagging. In this paper, we consider a simpler approach where the tag of the benchmarks do not change after the initial simulations (as discussed above).

To reduce the number of application-to-core mappings considered for a given combination of k cores, we consider simulating only those combination where there are more than $k/2$ matches between characteristics of cores and workloads. Through this exclusion rule, we are able to eliminate a large number of permutations thereby significantly lessening the simulation overhead while still doing a fair evaluation of every core combination.

Note that we can try SAHC or GA on top of above approach which will decrease number of instanced considered even more.

Tagging Cores and Benchmarks

We examined two techniques for tagging the cores in the library and the set of benchmarks used to evaluate each core. The first technique is based on core complexity while the second one involves tagging based on core parameters.

Following subsections will examine the methodologies.

1-tuple Tagging: Tagging Based on Complexity

This technique uses the notion of complexity (or complicatedness) of the cores and the benchmarks to tag them (tagging benchmarks is based on the notion of sensitivity to complexity of cores). The tag is a 1-tuple with values - *Simple*, *Moderate*, and *Complicated*.For cores, tags are rather intuitive. A core gets the tag *Simple* if it is relatively simple and small in terms of its various parameters. A core is assigned the tag *Complex* when it is relatively complex and big in terms of its parameters. The cores that not clearly *Simple* or *Complex* are marked as *Moderate*.

Benchmarks are tagged according to their performance on our library of cores. A few clearly *Simple*, *Moderate*, and *Complex* cores are picked and each benchmark is run on them. A benchmark that has more or less same performance (defined by a threshold) on a *Simple* core is tagged as *Simple*. If the performance difference is large, the benchmark is tagged *Complex*. If it is neither, the benchmark is tagged as *Moderate*.

The advantage of tagging according to complexity is that we can ignore details regarding the processor parameters and be able to categorize cores and benchmarks into somewhat simplistic categories. For example, it lets us put any type of core or benchmark into three categories which makes the machine learning-based search fairly simple in terms of matching. The limitation is that it lacks the details leading to inefficiency. For example, if a benchmark performs well on all cores that have high fetch width, the machine learning-based search will pick up the cores where not only fetch width is high but other parameters might be complicated as well.

k-tuple Tagging: Tagging Based on Parameters

Another technique that we examined is to tag cores according to their parameters. We categorized cores into 5 categories: Simple, D-cache intensive, I-Cache intensive, Execution units intensive, and Fetch Width intensive. Each tag, therefore, is a 5-tuple where each tuple is one of the fields. Multiple fields can be set at the same time. For example, a core with a large DCache and a large ICache is tagged as (0,1,1,0,0).

Benchmarks are tagged according to their improvements in performance when one of those parameters on a core running the benchmark became complex. For example, *Adpcmc* is execution unit intensive and also fetch width intensive, and is tagged as (0,0,0,1,1).

Note that a more effective tagging technique would be allowing each field in the tag to take parameter values instead of binary values. However, conceptually it is equivalent to the above technique with a larger number of fields per tuple.

The advantage of tagging cores according to parameters is that a lot of attention is paid to the specific parameter(s) that effect(s) the performance of a benchmark. For example, if a benchmark performs well only when the core on which it is being run has a high I-Cache size AND a large number of execution units, the machine learning-based search will be able to pick the right core just for such a benchmark.

A disadvantage of k-tuple tagging is that the search and matching becomes slightly more complex as multiple tags are allowed. In this paper, we declare a match when the characteristics of benchmarks are covered by the core's characteristics.

4 Magellan Framework

Figure 2 shows the overall structure of Magellan, a framework for multi-core design space exploration and optimization that we have developed, and the interface between the optimization techniques and the performance simulator. The optimization techniques refer to the search and machine learning techniques discussed in previous sections.

The inputs to Magellan are area, power, time constraints. A time constraint refers to the amount of exploration time

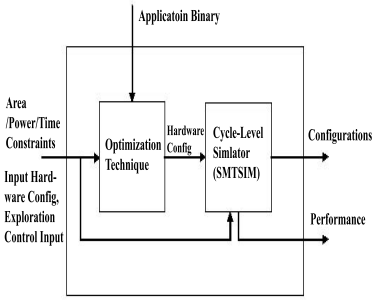

Figure 2. Overall structure of Magellan

that the designer is willing to expend. Magellan also receives as input a set of applications that the designer would want the processor to run effectively on. The output of Magellan is the hardware configuration of a processor optimized according to some objective function and the performance, area, and power characteristics of it.

Magellan can either be used as an automated tool multicore processor design exploration or it can be used as an aid to the experienced designer.

One fundamental limitation of Magellan is that it does not provide guaranteed minimum performance, not even probabilistically. Another limitation of the Magellan framework is that it currently takes as input only multiprogrammed workloads, not parallel workloads. However, this limitation is not fundamental and is due to the fact that it is not clear to the authors what is the best strategy for running parallel programs on heterogeneous architectures. Magellan also currently does not account for form factors and aspect ratios of the cores as well as the floorplanning issues, etc. when picking cores for a particular area budget.

5 Methodology

We make several simplifying assumptions to reduce the number of simulations we had to do for the paper. First, we assume that the performance of individual cores is separable, that is, that the performance of a four-core design, running four applications, is the sum (or the sum divided by a constant factor) of the individual cores running those applications in isolation. Second, we assume good static scheduling of threads to cores. Thus, the performance of four particular threads on four particular cores is the performance of the best static mapping. We also consider only major blocks to be configurable, and only consider discrete points. For example, we consider 4 cache configurations (per cache) (rather than all the intermediate values). But we consider only a single branch predictor, because the area/performance tradeoffs of different sizes had little effect in our experiments.

Modeling of CPU Cores

For all our studies in this paper, we model k-core multiprocessors (for k=4,6, and 8) assumed to be implemented in 0.10 micron, 1.2V technology. Each core on a multiprocessor, either homogeneous or heterogeneous, has a private L2 cache and each L2 bank has a corresponding memory controller. We base our processor microarchitecture model on the Alpha EV5 (21164). We evaluate 96 cores as possible building blocks for constructing the multiprocessors. This represents all possible distinct cores that can be constructed by changing the parameters listed in Table 1. The various values that were considered are listed in the table as well. Other parameters that are kept fixed for all the cores are also listed in Table 1. The various miss penalties and L2 cache access latencies for the simulated cores were determined using CACTI [6]. All evaluations are done for multiprocessors satisfying a given aggregate area and power budget for the k cores. We do not concern ourselves with the area and power consumption of anything other than the cores for this study.

To model the peak activity power and area consumption of each of the key structures in a processor core using a variety of techniques, we use a methodology identical to [4]. The cores represent a significant range in terms of power (4.72-12.98W) as well as area (3.45-14.38mm2).

Modeling Performance

All our evaluations are done for multiprogrammed workloads. We used twelve benchmarks used for constructing workloads. These benchmarks are randomly chosen from the SPEC2000 suite (*ammp, bzip, crafty, eon, mcf, twolf, mgrid, mesa*) as well as benchmarks from Olden (*deltablue*), IBS (*groff, gs*), and Mediabench (*adpcmc*) suites to ensure diversity. Every multiprocessor is evaluated on two classes of workloads. The *all different* class consists of all possible k-threaded combinations that can be constructed such that each of the k threads running at a time is different. The *all same* consists of all possible k-threaded combinations that can be constructed such that all the k threads running at a time are the same.

We find the single thread performance of each application on each core by simulating for 250 million cycles, after fast-forwarding non-representative instructions [1]. This represents 1152 simulations. Simulations use a modified version of SMTSIM [7]. Scripts are used to calculate the performance of the multiprocessors using these single-thread performance numbers.

6 Analysis and Results

In the following sections, we examine the effectiveness of the proposed machine learning-based techniques

Issue-width	1 , 2 , 4		
I-Cache	8KB-DM, 16KB-2way, 32KB-4way, 64KB-4way	L2 Cache	1MB/core, 4-way, 12cycle access
D-Cache	8KB-DM, 16KB-2way, 32KB-4way, 64KB-4way dual ported	Memory Channel	533MHz, doubly-pumped, RDRAM
FP-IntMul-ALU units.	1-1-2, 2-2-4	ITLB-DTLB	64, 28 entries

Table 1. Various Parameters and their possible values for configuration of the cores.

by comparing them against adapted exhaustive and intelligent search/optimization techniques. treating We also revisit quantitatively the various usage models for Magellan.

Treating Processor Design as a Search and a Machine Learning Problem

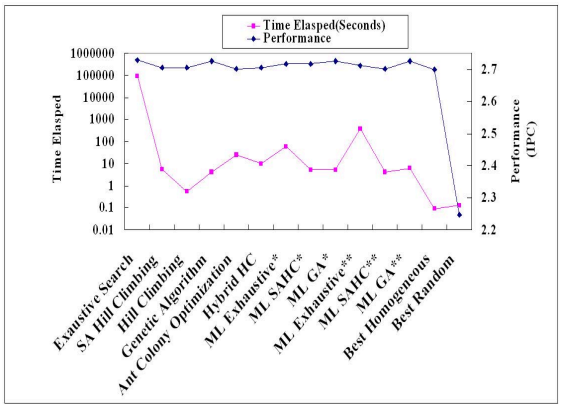

Figure 3. Throughput of the best 4-core multiprocessor discovered by the various techniques, and the time elapsed for each design space exploration. The results are for *all-different* workloads for an area budget of 45 mm^2 and a power budget of 60W. Single Asterisk indicates 1-tuple tagging and double asterisks indicate k-tuple tagging.

Figure 3 show the results in finding the best 4-core processor with our techniques compared against the *Exhaustive Search*, the *Best Homogeneous*, and the *Best Random* processor. *Best Homogeneous* is an exhaustive search over only homogeneous multi-core architectures. *Best Homogeneous* for four cores, for example, considers 96 different multi-core architectures (each corresponding to a different core type). *Random Best* corresponds to a randomly chosen multiprocessor that just satisfies the area and power constraint. Note that such a multiprocessor utilizes the available area and power well and hence would perform significantly better than a purely randomly chosen four-core chip multiprocessor.

The results show that the adapted search techniques as well as the proposed ML techniques perform exceedingly well. While exhaustive search indeed comes up with the highest performing chip multiprocessor, intelligent search/ML techniques discover processors that are within 0.1 percent of the "optimal" processor in terms of performance. In terms of the time taken for design space exploration, these search/ML techniques have orders of magnitude less overhead. Hill Climbing, for example, is over 168000 times faster than exhaustive search. Similarly, ML Exhaustive is 1600 times faster than exhaustive search. Genetic Algorithm emerges as the best search policy and performs within 0.1% of exhaustive search while being almost 23000 times faster. In fact, all the search/ML techniques are at least 3800 times faster than exhaustive search and perform no worse than 1% in terms of performance! To put these results into perspective, Random Best performs no better than 23% of the exhaustive search in spite of utilizing the area/power budget well.

The benefits of the proposed techniques become much more pronounced as the design space increases. Figure 4 shows the results. As we can see, the overhead of exhaustive search increases exponentially as number of cores increase. The overhead of our techniques, on the other hand, increase only superlinearly in the worst case, and only linearly in the best case. An interesting result also that ML when applied to intelligent search techniques, results in processors with comparable performance, but can often take significantly less time for exploration. For example, ML when applied to Steepest-Ascent Hill Climbing is over 81 times faster than the baseline Steepest-Ascent Hill Climbing.

As technology leads to increasing number of cores on the die and increasing number of parameters, exhaustive search may become impossible, and using one of the intelligent techniques may arguably the only way to do design space exploration.

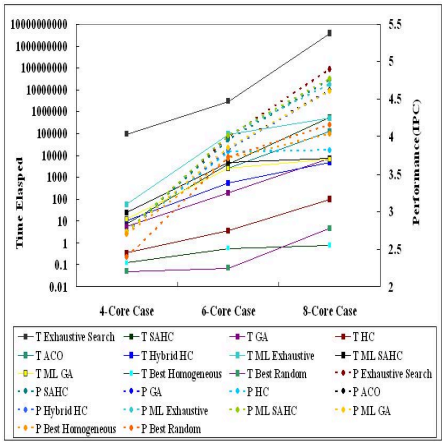

Figure 4. Throughput/time tradeoffs for various techniques for different number of cores for an area budget 50, 50

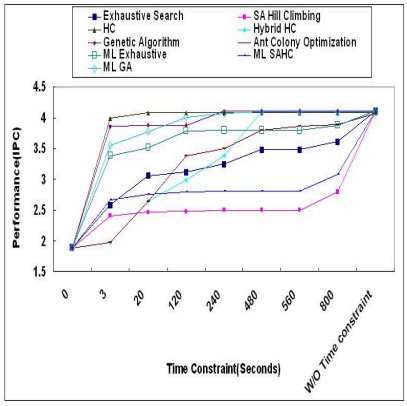

Figure 5. Tradeoffs for 6-core exploration for different fixed time budgets for an area budget of 80 mm^2 and a power budget of 70W

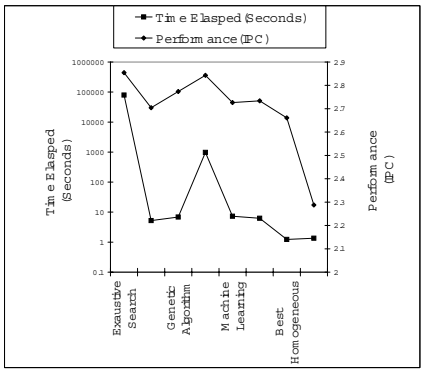

Figure 6. Throughput/time tradeoffs for various techniques for 4-core processor with L2 Cache Parametrized on budget of an area budget of 45 mm^2 and a power budget of 60W.

We studied two usage scenarios to demonstrate quantitatively the effectiveness of Magellan. First, we studied the effectiveness of using Magellan in terms of finding the highest performing 4-core processor in a fixed time budget. This may represent the case where a designer does not want to expend more than a certain amount of time to processor design space exploration. Figure 5 shows the results for *all different*. As expected, while some techniques evolve slowly towards a better processor (e.g., exhaustive, SAHC, etc.) due to unavoidable evaluation of bad/redundant processors, other techniques (e.g., ACO, HSHC, etc.) evolve faster as they try different starting points. Genetic Algorithm jumps to a high IPC the quickest as one of population might be already in a range of the "optimal" solution.

Figure 6 shows the results for the second usage scenario where our techniques are applied for multi-core optimization – core/L2 co-design, specifically. The goal is to again come with the highest performing 4-core chip multiprocessor, except that even L2 cache size is parametrized. I.e., the L2 cache can now be 512KB, 1MB, or 2MB, and depending on the size of the L2, other core resources may be constrained. As the graph shows, the search/ML techniques continue to perform significantly better than an exhaustive search.

7 Summary and Conclusions

We develop Magellan, a framework for doing fast and efficient multi-core design space exploration and optimization, that uses several well-known (and some less known) search /optimization techniques and two novel ML-based techniques to look through the multi-core processor design space. The intelligent search/ML techniques are at least 3800 times faster than exhaustive search and perform no worse than 1% in terms of performance! The proposed ML techniques perform up to 13% better than the adapted search techniques. These techniques also scale well with the number of cores while the overhead of exhaustive search increases exponentially. Magellan can be used either as an automated tool for exploration/optimization or it can be used to aid processor designers that prefer to rely on intuition by allowing fast refinements to an input design.

References

[1] http://www.cse.ucsd.edu/ calder/simpoint/.

[2] D. Fischer, J. Teich, M. Thies, and R.Weper. Efficient architecture/ compiler co-exploration for asips. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2002.

[3] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 195–206, New York, NY, USA, 2006. ACM.

[4] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 23–32, New York, NY, USA, 2006. ACM Press.

[5] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ASPLOS*, 2006.

[6] P. Shivakumar and N. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. In *Technical Report 2001/2, Compaq Computer Corporation*, Aug. 2001.

[7] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.