Simsys: a Performance Simulation Framework

José Noudohouenou
Vincent Palomares
William Jalby
Exascale Computing Research
jose.noudohouenou@exascale-computing.eu
vincent.palomares@exascale-computing.eu
william.jalby@exascale-computing.eu

David C. Wong
David J. Kuck
Jean Christophe Beyler
Intel Corporation
david.c.wong@intel.com
david.kuck@intel.com
jean.christophe.beyler@intel.com

ABSTRACT

HW/SW codesign or computer system purchase involves many tradeoffs, including the problem data size, choice of algorithm and compiler, types of HW subsystems used, clock frequencies of each, and number of cores. Simsys is a fast simulation tool set to examine various combinations of these choices, allowing specific HW/SW performance attributions. Simsys's measurement level and approach are keys to this operating speed and attribution.

A combination of modular tools forms Simsys's automatic procedure for system simulation and analysis. The paper overviews the tools and validates the proposed approach on 27 loop nest codelets extracted from Numerical Recipes. It also includes the experimental method and an error analysis.

Three performance quality metrics are defined and evaluated for two simple codelets, demonstrating several modes of performance failure and the weakness of intuition in detecting them, as well as illustrating how better tools could help lead to better computer systems. Future Simsys plans include model enhancement with more HW details and much more extensive experimentation.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques*;* I.6.5 [**Simulation and Modeling**]: Model Development – *Modeling methodologies*; I.6.6 [**Simulation and Modeling**]: Simulation Output Analysis

General Terms

Measurement, Performance, Design, Experimentation.

Keywords

Cape Modeling, Frequency Scaling, Performance Projection, Stability, Scalability, Speedup, Saturation

1. INTRODUCTION

Simsys has several simulation purposes and benefits. It can be used for system configuration or design at several levels of resolution, it can pinpoint HW/SW performance details, and it is fast. On the first point, for end-user system configuration, Simsys HW nodes could be cpu, memory, disk, network; for OEM design studies, it could be used by microprocessor designers to analyze interactions of scalar, vector floating-point, and L1, L2, L3, RAM tradeoffs, or it could be applied to entire SoCs.

Conference'10, Month 1–2, 2010, City, State, Country.

Relative to the second point, the Simsys process is based on microbenchmarks, small program segments called codelets, and their effects on the HW nodes under consideration. This provides feedback at the HW level as well as algorithm/application levels, and could lead to problems with instruction types or the compiler in improving runtime libraries or whole applications, for example. On the third point, Simsys's speed allows many whole-application studies, including multiple data sizes and program execution paths as a complement to existing simulation and analytic methods.

Because of its global scope across HW nodes and many applications, otherwise impossible design space exploration tradeoffs may be feasible using Simsys. Furthermore, program tuning may be aided by frequency variation studies that can filter candidates by their performance scalability potential, e.g. poorly scaling codelets need improvement for use on a faster system.

The most costly business errors in computing occur in two related areas: producing the wrong systems for a given market segment, and buying the wrong system for the work at hand. These problems have largely driven the history of computing companies' successes and failures. Specific errors arise from weak performance, misunderstanding the details of market-specific applications and data set requirements, energy consumption, system cost, etc. This paper describes some details of how Simsys can help on the performance and application analysis fronts. For system designers, pre-silicon analysis of whole workloads is a key need. For system buyers, analysis of given HW choices for problem-type (data size) variations on an applications set is important. The Simsys approach has the speed and flexibility to target both areas.

The Simsys framework is based on a chain of tools that perform independent tasks. This provides a modularity that will allow it to grow in functionality. For example, HW performance counters are being added to its current purely SW approach to pinpointing performance problems. This should provide more data in some cases, and also allows error checking by comparing one method with the other. Of course, when one tool in the chain is too slow or lacks flexibility of use, it can be replaced with another.

The paper's contributions include:

- Simsys to quickly explore design space allowing HW node change, frequency and core count variation, and provide insight about performance and node saturation.
- SW characterization by a set of codelets that represent whole applications. Numerical algorithms covering four key areas of Numerical Recipes (NR) are demonstrated.
- Metrics to evaluate design quality globally across HW nodes and SW codelets are discussed and used in examples, varying data size as well as HW variables.

Section 2, explains what Simsys provides and why it is important. Section 3 details its infrastructure as a combination of other tools.

Section 4 validates the methodology by presenting experimental results, including characterizations of NR codelets and Simsys results. The paper concludes after Section 5's related works.

2. MOTIVATION

The approach advocated in this paper is intended to be useful eventually for the full range of applications. Numerical Recipes [6] was chosen as a starting point because it provides a large, well-respected collection of routines that represent a wide variety of HPC applications and has several appealing properties.

The source code is naïve, i.e. not written or tuned for performance on any particular architecture. In future studies, the results below can be contrasted with optimized forms of the same algorithms. The basic ideas will remain the same, but for the same algorithms, an improved source code or compiler will yield improved performance, which Simsys can reveal. The Simsys NR results are a prototype of its use for general software improvement.

The source code has less than 200 lines per recipe, making it easy to deal with experimentally, but its rich diversity of source (and assembly) instructions yields a variety of simulation performance results. Any data size can be run for each recipe, allowing the full range of performance behavior to be observed; this is a necessity if simulation runs are to expose real world system behavior.

Simsys's important techniques and benefits follow.

Measurement techniques:

1. Measurement primarily uses SW techniques. Simsys uses HW counters only when needed, as they often fail to deliver meaningful results in user-specified areas. The SW approach provides flexibility in targeting specific topics. Time and node saturation information can indicate specific HW and SW improvements needed.
2. Simsys can focus on any codelet sets that are used in various real applications. This provides understandable SW details, which can provide wider benefits than are possible using only industry-standard benchmarks.
3. Simsys measures performance at any node frequency for unicore and multicore systems and projects performance to new systems with new frequency ranges.
4. Simsys varies data sizes to understand applications in many usage settings. This is also impossible with most industry-standard benchmark analyses.

Modeling benefits:

A. Simsys's simulations are fast, yet comprehensive in code and data set coverage. By combining information from all HW nodes, they provide a system level view of potential performance gains beyond the results of microprocessor-only simulations.
B. As new nodes are needed, and new measurements are available, the model is easily extensible to include them.
C. Simsys measurement and modeling introduce various errors (see below). But by making major improvements to the workload scope possible, it avoids many of the omission inaccuracies in traditional simulation studies.
D. Using Simsys for SW performance studies to show sensitivity to frequency change and saturation patterns can help developers understand which codelets offer promise, with more accuracy than profilers allow.
E. The techniques are derived from first principles and are directly translatable back to real HW/SW inputs. This avoids some of the problems engendered in statistical and heuristic modeling methods.

Three performance *quality metrics Q* can be analyzed via Simsys:

- Stability (*cores, freq, perf range*, ***D*** or ***codelets***)
- Scalability (*cores*, ***freq***, *D*)
- Speedup (***cores***, *freq*, *D*)

where the bold variable indicates the main purpose of each metric. The first measures how well performance stays within some range for a given data size range (*D*) on one codelet, or across codelets. Good stability prevents performance surprises at runtime. The second is how well performance scales as frequency is increased (e.g. memory-bound apps do not improve if only cpu frequency increases). This is indicative of performance potential on a faster (or slower) system. Speedup is limited in this paper to throughput impacts of a memory hierarchy. Measuring thread interactions can capture thread parallelism (e.g. OpenMP run-time library effects).

Scalability and speedup both have natural target values for a given system: the system's frequency ratio and number of cores, respectively. Stability's specific *performance range* has a domain-specific nature ([1] gives some specific measurements and examples). Users of various types of applications usually expect a performance "sweet spot" for a particular range of data sizes, and perhaps an expected core-count range. All 3 metrics are important in evaluating current computer systems and Simsys is applicable to all three; examples will be given in Section 4.

Figure 1 is a Simsys example showing the potential performance gains (in machine cycles per loop iteration) of a faster unicore system, using data from the lowest cpu frequency (LF) run to predict performance at the highest frequency (HF=2X LF), and vice versa. The results show that the tridag2 codelet can benefit from a frequency increase for data sizes less than 100K; for larger data the system is memory bound and since memory frequency is constant, no performance benefits occur. Thus, the types of problems to be run should dictate the answer to a new purchase.

In terms of accuracy, Simsys gives 3.5% maximum difference (one data point <7%) between the simulation and measurements when using low frequency to predict high frequency, or vice versa.. Simsys can be useful for considering faster hardware to increase performance or to reduce overall costs or energy consumption. Both unicore and 4-core cases will be discussed.

The overall planning and codesign of systems is a funnel process requiring several years. The input is a set of market ideas and the output is a microprocessor and system to serve a wide market well. Errors arise at each step in the process, and the goal of the Simsys approach is to reduce these errors by replacing intuition by more analysis. At the funnel's front, Simsys can deal with codelets extracted from many applications, because of its speed using the Cape-sim methods. Deeper in the funnel, Cape-cod methods [2,3] can find a set of *optimal* designs satisfying global constraints. Cape-sim can be used to evaluate a *single* design idea on many applications, isolating poorly performing codelets. These may be improved by better ISA, system nodes, compilation or algorithms, or if that is impractical, can form the basis for a *second* design.

3. TOOL CHAIN

The following is an overview of the Simsys framework. Simsys provides insights about performance vs. data size (*D*), directly attributing saturation to the cpu or memory hierarchy. It also gives comparisons of frequency change for individual system nodes and system parallelism degree. Finally, its speed allows the evaluation of metrics for individual programs on many architectures, as well as the comparison of many programs on a given architecture.

Codelet Finder [4], Decan [5], the MicroPerf tool set [6], and the Cape simulation tool [3] all compose the Simsys tool. Figure 2 provides a visual description of the Simsys system. As input, Simsys accepts applications and multiple data sets. Tools and benchmarks that use only one data set may lead to biased profiles, as well as tuning and optimization mis-directions. Experiments with Simsys demonstrate this, and are summarized in Section 4.

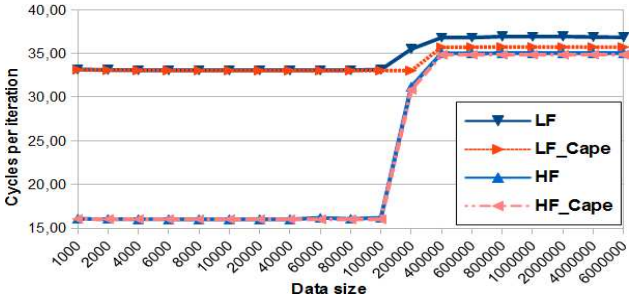

Figure 1 Simsys performance vs. data size & frequency

Simsys's first step is Codelet Finder, an automatic source code tool to extract loop nests and create stand-alone codelets containing: the original loop nest, a wrapper, and the application's context at the loop's entry. It supports sequential and parallel C and Fortran based applications and well-known benchmark suites, e.g. NAS and SPEC. Codelet Finder allows focus on each loop nest separately, and using the original context to simulate the application's original behavior with only the extracted codelets, which contain the most time-consuming loops per application.

Simsys's second step is Decan, a static binary rewriting tool, which transforms a binary file's loop nests into several variants and creates new binary executables. Decan can perform many transformations such as removing every SSE/AVX Load/Store instruction or FP instruction, inserting new instructions, and instrumenting the code. Most variants modify the semantics of the original loop nest, but Decan's purpose is to study the loop nest's performance and not the numerical result itself. Simsys uses Decan to produce two main variants: the loop nest without any SSE/AVX Load/Store instructions (fpi) and the loop nest without any SSE/AVX FP operations (lsi).

To calculate the impact of processor frequency changes, Simsys requires further code decomposition. Load/Store instructions are divided into groups: a group is a set of Load/Store instructions referencing nearby addresses: i.e. addresses using the same index and base registers values but with different offsets. Load/Store instructions belonging to the same group target the same array/data structure. Key parameters of a group are: number and type of Load/Store instructions, stride, operand location (L1/L2/L3/RAM). There is a large amount of reuse of each group between different codelets. This allows building a database containing all of the performance information relative to all groups. To achieve that, MicroPerf tool set creates synthetic microbenchmark programs to evaluate both group performance and potential interaction (overlap) between groups. If the memory system is saturated, original execution time is the *sum* of individual group times; otherwise the two groups overlap and performance is the *maximum* value. For arithmetic instructions similar microbenchmarks are also generated.

After Simsys creates the binaries, whether from MicroPerf or Decan, the tool executes the binaries on target hardware. Decan executes the binaries using the given input data sets on *initial* hardware at *initial* frequency only. Decan automatically inserts the instrumentation required to time the loops using the TSC register, providing high resolution, low overhead timing. Microbenchmarks use MicroPerf's second tool MicroLauncher, which executes the micro-benchmarks in a closed environment on both *initial* and *final* hardware at *initial* and *final* frequencies. The microbenchmarks are enclosed in a repetition loop to reach steady state behavior. A script is used to execute each program, providing statistically sufficient data to calculate mean values and overcome the time fluctuations inherent in individual runs.

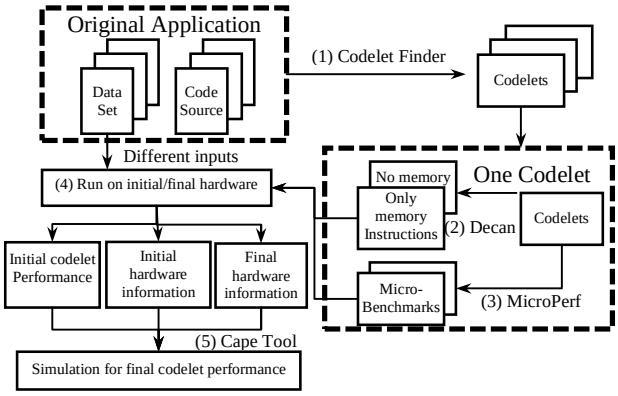

Figure 2: The Simsys Tool Chain

Finally, the Decan binaries are measured to provide the *initial* codelet performance, and the micro-benchmarks are executed at the *initial* frequency and the *final* frequency, and stored in the database. Once Simsys finishes the multiple experiments, sufficient data is available for the Cape tool, which uses the data produced by Decan variants and MicroPerf benchmarks to compute performance changes.

Cape Definitions

The Cape-sim technology used in Simsys is based on the full Cape-cod codesign tool [3,2]. It uses initial codelet performance and hardware information to create an initial performance model and projects the final codelet performance based on the created model and final hardware parameters supplied. A simplified description of the process is given below.

In the Cape-sim prototype, performance is expressed in terms of *time t*, or *rate* via node *computational capacity C* which represents the amount of *bandwidth B* used in a computation. Both are measured in [b/s] or [ops/s] ($C=O/t$, using O[ops]). Summing appropriately over HW *nodes* and SW *phases*, allows the expression of *performance* [b/s or time] and system *initial cost* [total B], of whole or partial systems and computations. For any HW node i being used for a duration of T_i, by a computation taking time T, *relative saturation* σ_i^C is defined in Eq. 1. and, Eq. 2 defines C_i using σ_i^C and BW B_i. For HW node i, $0 \le C_i \le B_i$, and for two HW nodes, i and j, Eq. 3 defines $\mu_{i,j}$, an *invariant ratio* across BW changes, for fixed SW. For HW node i, $0 \le C_i \le B_i$, and a computation taking time T using the node i for a duration of T_i, Eq. 4 defines O_i, another invariant quantity across BW changes, for fixed SW.

In terms of these, system-wide performance codesign problems can be formulated. First, SW is measured on a system with known physical parameters; since the SW does not change from one simulation to another, both $\mu_{i,j}$ values of Eq. 1 and O_i values of

Eq. 4 are invariant. By linking *local* equations for each node during each computational phase, to *global* equations spanning all phases, complete system behavior can be described. Evaluating

large systems of equations for various parameter values provides

$$0 \le \sigma_i^C = \frac{T_i}{T} \le 1 \quad (1) \qquad C_i = \sigma_i^C B_i \quad (2)$$

$$\mu_{i,j} = \frac{C_j}{C_i} = \frac{1}{\mu_{j,i}} \quad (3) \qquad O_i = C_i T = B_i T_i \quad (4)$$

for fast system *simulation*, and performance results. With a linear model, as a system moves from one computational phase to another, HW node saturation changes, and the system operating point makes state transitions.

Using Cape-sim, complete systems can be *simulated* with mixed fidelity by choosing physical values for *B*, to obtain performance for given initial cost. These initial measurements could be obtained by running the SW on a real system and/or lower level simulators (e.g. register transfer level). The simulation results can be regarded as time varying at the level of <*node*, *phase*> resolution, or global by summations over nodes and phases. To simulate new system designs, new physical values for B are given. For an $\langle \#node, \#phase \rangle$ simulation, using Eq. 2 with initial C and new B, new relative saturations can be computed for each node, requiring time proportional to $\#node$ operations.

Cape-sim looks for the node with largest σ^C and determines the new capacity to make $\sigma^C = 1$ (i.e. fully saturated), also requiring $\#node$ operations. The new capacities of other nodes can then be computed using Eq. 3, taking $\#node - 1$ operations. Finally, the new time can be computed using Eq. 4, taking a constant number of operations. So, Cape-sim runs in time proportional to $3 \times \#node$ operations. The same procedure is done for each phase, so an $\langle \#node, \#phase \rangle$ simulation requires time proportional to $3 \times \#node \times \#phase$ to finish. Since the equations are linear, each operation can be done efficiently. Empirically, we have simulated 80K simple scenarios within 1 minute. Problems of accuracy arise due to measurement errors, and preliminary Cape results are given in Section 4. Error analysis is also performed, relating simulation errors and measured variations in μ and O.

In general, Cape-cod uses a system of equations based on the above variables to formulate codesign problems using constraints and optimization objectives. When such problems are linear they can be solved by linear programming; cases where nonlinearities arise can be optimized using geometric programming. But Simsys uses only part of Cape-cod, and is linear in parameter count (mostly matrix-vector ops). Finally, after Simsys performs all the previous steps, the tool generates information about performance changes as node frequency and *D* are varied for target hardware.

Modeling Limitations

The most time consuming step of Simsys is generating reference data, which has 3 major factors:

1) Steady state execution of the target loop is obtained by surrounding it with a repetition loop of, say, 100 iterations.
2) Measurements for all of the variants (cpu, levels of memory hierarchy, etc.) may require 10 variants of the original code
3) To avoid machine and OS variability every experiment may be rerun up to 50 times to get statistically meaningful data.

As the Cape time is relatively negligible, if the running time for a given codelet is *T*, total Simsys time is less than a 100 x 10 x 50 x T = 50,000 slowdown relative to real time. These are conservative numbers, but even without tuning or refinement are several orders of magnitude faster than typical cycle-accurate simulators.

The speed of Simsys would allow its use for design space exploration, in conjunction with traditional simulators to examine the details of specific topics. Since computer design projects usually focus on a half-dozen complex issues, design teams must step through a series of design studies focused on one issue at a time. Integrating the results leads to difficult compromises that must be made intuitively, because current simulation facilities are too slow to look at many global HW/SW tradeoffs. The results always lead to surprises in the marketplace. Some of these could be moved upstream using Simsys-like tools, and in some cases remedied by pre-silicon HW and SW modifications or market refocus, as in the funnel discussion of Section 2.

System design requires exploring many degrees of freedom – change in ISA, memory hierarchy, cpu, I/O, network, etc. Current Cape-sim applicability is limited to simulating variations of a given architecture (ISA and node types), such that microbenchmarking and Decan transformations are applicable to both architectures. Since binary instructions are analyzed, a new instruction type could be included if (estimated or measured) performance data were available. Architectural-change performance differences could be estimated if data were available from a simulator or analytical model. In general, measurement accuracy determines applicability. We are actively extending the range of applicability, including error analysis and reduction.

4. EXPERIMENTAL RESULTS

The following section presents Simsys's validation. We use Simsys to project codelet performance with respect to frequency changes. Experiments have been done for both initial and final frequencies. To validate Simsys, we use initial frequency performances to project final frequency performances and compare the projected results against measured ones.

The test machine was a 3.30GHz quad-core Xeon E31240 with 12GB of RAM. The operating system is Linux with a kernel version 2.6.32-5-amd64. Finally, the compiler used was ICC version 12.1.3 with the -O3 -xSSE4.2 optimization flags.

We chose four important algorithm types from Numerical Recipe [7] codes; see Ch. 2. Solution of Linear Algebraic Equations, Ch. 11. Eigensystems, Ch. 12. Fast Fourier Transform, and Ch. 14. Partial Differential Equations. From these chapters we selected 17 important recipes, covering much of the functionality of all those chapters. Codelet Finder automatically extracted 96 codelets from these 17 algorithms. Individual recipes yielded up to 26 distinct codelet types, but most have less than 10. By profiler analysis, we concluded that the 27 codelets discussed below, represent most of the execution time in the 17 key numerical recipes.

The codelets used have loops of varying complexity and include single-, double-, and mixed precision data. Scalar, vector-vector, matrix-vector, and matrix-matrix type algorithms are present in various combinations, as shown in Figs. 3 and 4. They contain potential barriers to fast execution including reduction/recurrence statements, scalar instead of vector instruction use, and non-stride 1 memory access. In fact, only 5 of the 27 have none of these difficulties; of these, 4 are memory saturated and one is cpu saturated. Some of the problem codelets contain other problems, including division operations, single-double precision conversion overhead, etc. Certain common problems, e.g. alignment, can be eliminated in preprocessing steps [3], or dealt with directly using appropriate microbenchmarks.

All codelets were executed for 19 data sets. The 1D arrays were executed by sweeping the range of 1K to 6M elements. For 2D arrays, a similar number of elements were accessed by stepping through array sizes from 100 to 2K. This flexibility during simulation provides much more insight than is possible using most industry-standard benchmarks (including NAS and SPEC),

which allow at most, limited data set variation (e.g. 2 or 3 data sets). And of course, Simsys may be applied to new applications as they arise without waiting for committee agreement on new benchmark suite releases.

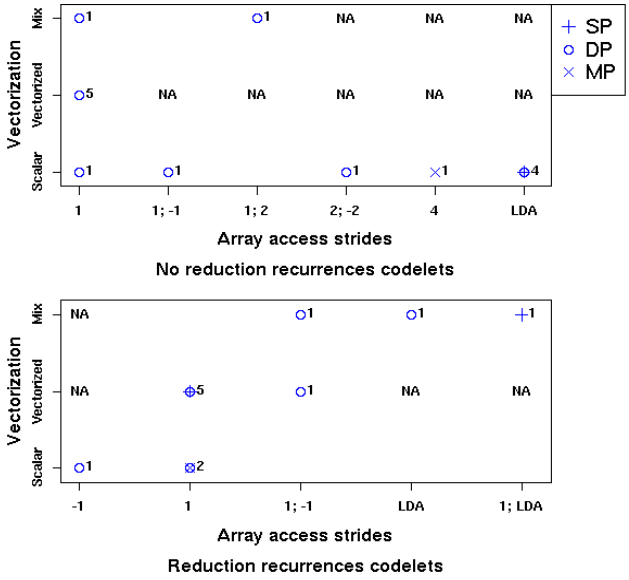

Fig. 3 Codelet vector/stride characteristics. Each point represents a codelet using either Single Precision (SP), Double Precision (DP), or Mixed single and double Precisions (MP) data with various access strides including LDA as short-hand for Leading Dimension of an Array. NA refers to impossible cases, e.g. non-unit stride is not vectorized. The number located on the right of each point denotes the number of piled up codelets, which have identical characteristics.

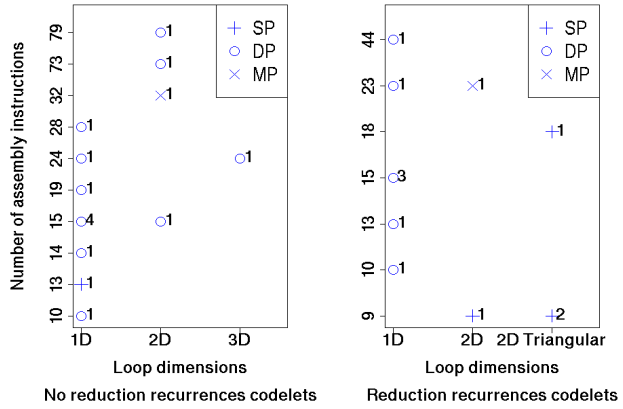

Fig. 4 Codelet loop characteristics

Saturation Patterns

As the previous sections explained, Simsys currently provides users with bottleneck information. Decan generates two important code variants: the floating-point variant (fpi) without any SSE/AVX load/store operations and the memory variant (lsi) without any floating-point arithmetic instructions. Figures 5 and 6 show cycles per binary iteration vs. data sizes, for two codelets, with four performance curves per figure: the original code, the floating-point and memory variants, and the predicted values for the full code. The predicted values for Figures 5 (resp. 6) each consider as a base point the highest frequency, and predict the performance when decreasing (resp. increasing) the frequency.

Fig. 5 corresponds to a vector divide, run unicore, low frequency.

The original code and the floating-point variant remain at 85 cycles for all data sizes, showing floating-point saturation. The memory version contains 3 plateaus: corresponding to data in the L2, L3, and RAM levels. The lsi variant's cycle count grows from 10 to almost 50 cycles, but never reaches the original code's cycle number, so does not have any impact on overall performance because the original code's cycle count is higher, i.e. memory instructions are not the bottleneck for this code. Note that Simsys correctly determines the global performance and correctly predicts the cycle counts within 3% error.

Fig. 6 corresponds to the same code as Fig. 5, but this time in a 4 core version and high frequency. For data sizes larger than 10 000, the code is clearly data access bound while for smaller data size, the code is floating-point bound. Again Simsys gives very good prediction within less than 3% error.

In both figures, the cycle count for the lsi variant increases as *D* increases, because as the data no longer fits in a given memory hierarchy level, the higher latency of the next level starts dominating the code. Fig. 6 shows that (unlike Fig. 5) increasing *D* results in memory saturation because with more cores, additional memory conflicts reduce effective BW. Furthermore, a higher frequency core effectively reduces the memory BW so the memory is relatively much slower for a given *D* value.

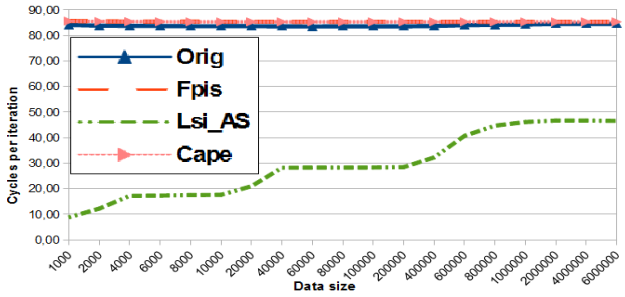

Figure 5 FPI bound loop , unicore low frequency

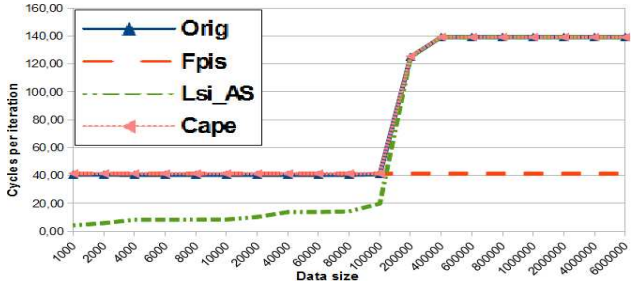

Figure 6 Saturation transition, 4-core, high frequency

Saturation Types

An ideal codelet extractor would find codelets that cover real applications well and saturate a single HW node throughout each codelet's execution. Codelet Finder extracts naturally occurring loop nests, not always yielding single-node saturation. Two nodes allow four possible saturation cases; either a single node, both nodes, or some other node not being modeled is saturated. When both are saturated, saturation alternates between the nodes within the execution of a single codelet. In this case, Cape does further analysis to determine how much of the time each single node is saturated, and how much time both are simultaneously saturated.

Realistically, more than two nodes may be needed for modeling.

For example, disk and network nodes may be needed in general (but not for NR applications). However, any application may occasionally saturate the instruction handling or control unit of a system. Finally, both the cpu and memory hierarchy should be considered to contain lower-level nodes, e.g. L1, L2, L3 cache and RAM, to provide more HW design or SW tuning information.

In the current form of Simsys, a number of these issues are dealt with, but will not be discussed in detail here. We have developed the R method based on Eqs. 1-4, to deal with cases where no node appears to be saturated. Also the memory hierarchy is broken into the 4 levels mentioned above, by using microbenchmarks and memory access grouping tests. Storage unit size should be used in addition to BW, and work is underway to include this. Finally, as a check and enhancement of the methods described in this paper, Simsys is incorporating use of the HW performance counters when they are reliable and can enhance Simsys (e.g. to validate cache level analysis by counting misses per level).

Figure 7 shows how often floating-point and memory saturation define the original execution time in the 27 NR codelets, with results for 1-core versus 4-core execution and low frequency versus high frequency. There is a data point for each data size per codelet, so for *n* data sizes, saturation is accounted for *n* times. If the codelet is always memory saturated, the mem histogram augments by *n* counts, but if it is only saturated by memory for the largest *m* data sizes, the cpu histogram increases by *n* - *m* points and the mem by *m*.

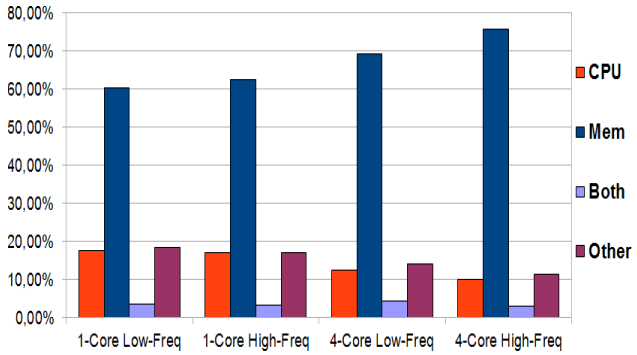

Figure 7: Saturation ratios for the 27 Numerical Recipes

Each configuration has four histograms providing percentages of data points saturated by cpu, memory, both, or by another HW component. Figure 7, represents 513 cases due to 19 different data sizes for 27 codes, per core and frequency configuration. The results show that *cpu* and *mem* saturations represent a majority of the cases. On 1-core with low frequency, there is 18% of *other* HW component saturation, 18% are *cpu* and 60% are *mem* saturated. However, at high frequency with 4-core execution, the *mem* saturated codes reach 75% and the *other* hardware component group drops to 11%.

Higher frequency and more cores stress the memory subsystem, resulting in the saturation of formerly non-memory saturated codes. Fig. 7 shows that in relatively few cases, both *cpu* and *mem* saturate part of the time, on multiple cores and high frequencies.

Performance Projection Errors

Tables 1 and 2 present accuracy data to validate Simsys's simulation aspect and ability to detect saturation changes. Each of the 27 NR codelets was run for 19 data sets varying in total size by more than 3 orders of magnitude to sweep out memory hierarchy behaviors. The columns show the % errors in Simsys frequency change projections of floating-point and memory hierarchy performance for unicore and 4-core systems, using both capacity (rate) and time performance metrics. Runs on 4-core systems use 4 copies of the same codelet at once, simulating the throughput of a system; the 4 core systems use the same shared system memory (L3, RAM) as used by the unicore runs. Many of the remaining errors occur at crossover points from one saturation regime to the other – at such points instruction processing can cause problems, e.g. in the reorder buffer (ROB). A few errors in plots like Fig. 5 and 6 may be easily tolerable when studying the global performance of many diverse applications.

The errors summarized in these tables compute an accuracy as the absolute difference between measured and predicted performance time divided by the measured time. Percentages are generated by counting the total number of Cape results that exceed 5% (Table 1) and 10 % (Table 2) error thresholds, based on a total of #codelets per type X 19 data sets X 2 frequency changes (up and down) = #codelets per type x 38. Note that the time column may

% cases with errors > 5%					
unicore			4 core		
fpi	lsi	time	Fpi	lsi	time
16%	*3%*	*10%*	*10%*	*2%*	*11%*

Table 1 Error-summary of 27 NR codelets (5% cases)

% cases with errors > 10%					
unicore			4 core		
fpi	lsi	time	fpi	lsi	time
8%	*1%*	*4%*	*5%*	*1%*	*4%*

Table 2 Error-summary of 27 NR codelets (10% cases)

be correlated with the fpi/lsi column because the saturated node capacities (fpi or lsi), are used to compute time.

Currently, Simsys includes a number of error tests that are used to detect errors and help improve accuracy. Modeling more node types (cache levels, instruction processing unit, etc.) is expected to reduce errors and provide more architectural insight. This section confirms Simsys's code decomposition by showing how Decan-based variants of memory and floating-point are important in predicting performance when varying frequency or data size. Few codelets have another HW-saturating node for the 27 math library codelets. Microbenchmark use to determine performance change is validated by accuracies achieved. This evaluation illustrates performance simulation of varying frequency or data size, to analyze which hardware component to optimize for a desired performance boost, etc.

Quality Metric Examples

Consider two simple codelets Hqr13 and Svdcmp11, as well as their performance in Figs. 8 and 9. These codelets were extracted from a deeper loop nest and parameterized to simulate execution of the innermost loop. They are among the simplest of the 27 codelets being discussed, but their simplicity adds to the impact of the difficulty in understanding the following *Q* (recall Section 2 quality metrics) behavior as *D* varies. Each codelet has a syntactic issue that could cause performance problems. These problems are typical of the performance difficulties in this codelet set.

Hqr13 has a simple reduction on *s*. Svdcmp11 is indexed "incorrectly" for columnwise Fortran storage so it has non-stride-1 storage access, and will not vectorize as is. How these two will compare on the three quality metrics for this architecture is

intuitively hard to guess. Simsys can reveal the details as follows.

```
Hqr13 :SUBROUTINE codelet (n, m, i, a, res) ! m = 10, i = 1
        do j = 1, n
                s = s + abs (a (j, i))
        end do
```

```
Svdcmp11 SUBROUTINE codelet (n, m, i, a, f) ! m = 20, i = 1
        do j = 1, n
                a (i, j) = a (i, j) * f
        end do
```

Figs. 8 and 9 show performance for 4 architectures (1- and 4-core, low and high frequency) across the full *D* range. A band from *D* = 40K to 400K is chosen to study the three performance metrics of Section 2. Table 3 summarizes the quality results of these two codelets, where the *stability* goal is 0.5, i.e. *fastest cycles per iteration* /*slowest cycles per iteration* > 0.5 is the stability goal.

The first two columns show stability for each of the codelets across all 4 architectures, and $40K \le D \le 400K$. The 3rd column compares the stability of the two codelets on single architectures at *D* = 40K. The results shown are the worst cases of high and low frequency measurements. The frequency *scaling* expectation is 2X, and the core *speedup* goal is 4X.

Hqr13 has good unicore stability but weak 4-core stability, as well as good scalability and speedup, except at *D* = 400K. The latter problem follows from the conflict-driven memory time increase for 4 cores that usually occurs for multicore systems, due to additional memory traffic. A potential fix would be to increase L3 or block the loop for better cache reuse.

Svdcmp11 has quality issues with all 3 metrics except 4-core stability. Reindexing (e.g. loop interchange) would improve this. These both can be regarded as poorly formulated benchmarks for a codesign simulation, but in a performance tuning situation, the SW restructuring needed would be immediately clear to an expert. To analyze different markets for this system, the *D* range can be varied further. The relative stability of the two codelets is shown

***Q* metric**	**Cores**	**Hqr13**	**Svd11**	**Relative**
		$40K \le D \le 400K$		Dmin=40K
Stability (*cores*, *freq*, *perf range*, ***D*** or ***codelets***) high and low frequency	1	>.94	< *.42*	.5, Dmin
	4	< *.42*	~1	.06, Dmin
		Bad *D* points		
Scalability (*cores*, ***freq***, *D*) >1.67, all freq	1	None	>80K	
	4	400K	all bad	
Speedup (4 ***cores***, *freq*, *D*) > 2; all freq		400K	all bad	

Table 3 Quality Results

to be good for unicore and poor for 4-core systems at *D* = 40K.

By formulating *Q* threshold ranges, this analysis can be extended to give clear expressions of *Q* limits for complete application workloads for a given system. The same approach can be used to study the *Q* effectiveness of a set of computer systems (existing or proposed) for each individual program or for the entire suite. The approach also allows evaluation of the modification of particular programs for better *Q* on particular systems.

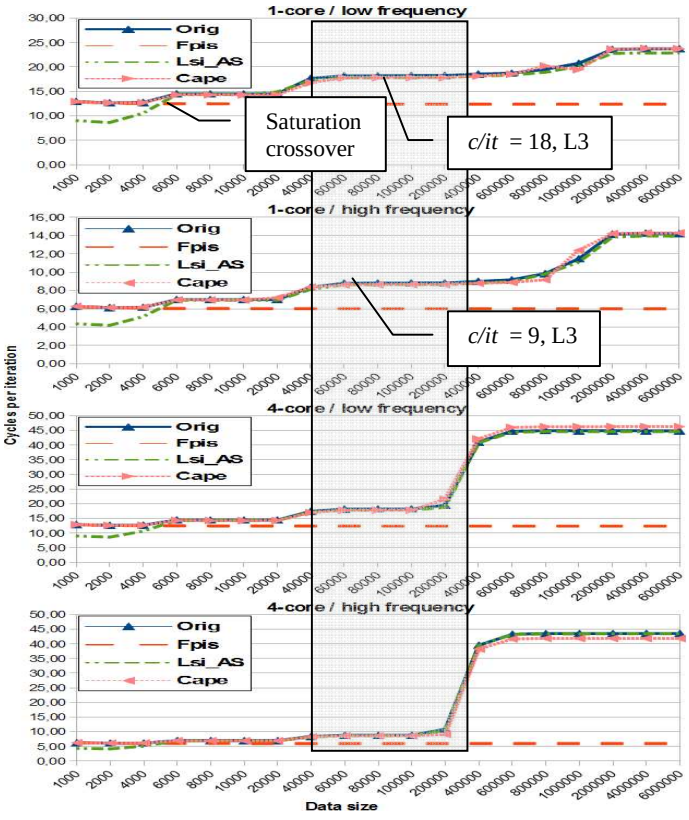

Fig. 8 Codelet 1: Hqr13 Performance vs. *D*

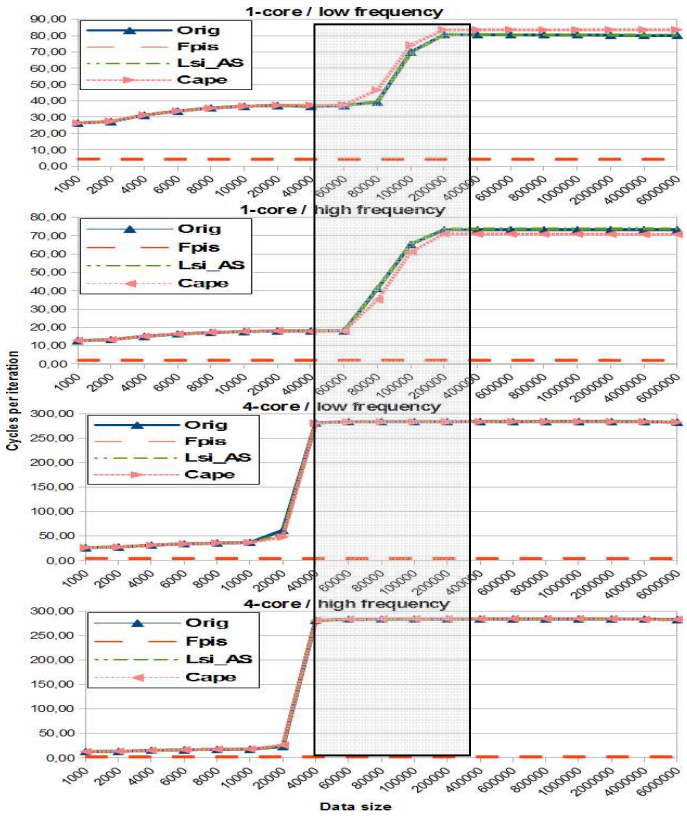

Fig. 9 Codelet 2: Svdcmp 11 Performance vs. *D*

Results like those of Table 3 are made possible by the comprehensive and global nature of Simsys's simulation scope. The stability performance range is market dependent, and using this approach can provide a good understanding of the applicability of a given system to particular real-world workloads.

5. RELATED WORK

Realizing the importance of HPC, many researchers have focused on predicting and simulating numerical program performance. In [8], a predictive analytical model uses as input, basic machine performance numbers, e.g. latency, bandwidth and application characteristics to represent current systems' performance, and reveals bottlenecks to focus tuning efforts. The model considers full applications analysis but does not examine co-design issues for machine upgrade, whereas Simsys demonstrates the benefits of code and machine tuning.

In [9] , the authors consider factors affecting code performance and scalability before developing a framework for performance modeling and prediction. The proposed framework combines tools and simulators to gather profiles and application signatures, and provide automated prediction methods. Although this is useful for performance analysis in several dimensions, the authors measure machine memory bandwidths using a benchmark derived from the STREAM benchmark [10] and combine memory hierarchy rates with application signatures to predict memory execution times. The work does not discuss codelet saturation (to better understand system performance).

Simulation tools often provide users with a single reference point for specific hardware parameters. The CAPE simulation tool uses a mathematical model; a number of related modeling approaches have been proposed [11,12,13,14]. Among these works, [12,14,13] simulate instructions one by one, so the simulation time is proportional to the duration of a workload. [11] is an analytical model to express interactions between processor and memory. They focus on determining IPC, using performance characteristics of the whole program. Simsys projects more fine grain details like specific node saturation for a part of a program. Simics [13] is a full system simulator that can run a real OS and needs a detailed timing model for timing simulations. Simsys does not require a user provided timing model, because Decan breaks down the codelet automatically and the timing is measured empirically. Rsim [14] is a multiprocessor simulator that models details about interprocessor events, which can be time consuming. Simsys can simulate a multiprocessor efficiently by modeling individual processors explicitly. Shared resource (e.g. memory) contention will be modeled by having a shared saturated node. Asim [12] is a modular simulation framework to ease simulator development by letting simulator developers focus on smaller simulation modules. In addition to aiming at improving simulation speed, [12] aims at speeding up simulator development.

6. CONCLUSIONS AND EXTENSIONS

Simsys can answer "what if" questions, e.g. Would the performance increase if the processor were faster? How would throughput increase with a multiprocessor? How does application data size affect execution time? The experiments show Simsys's accuracy in predicting performance when modifying the frequency or data sizes for 27 Numerical Recipe codelets, covering typical high performance and mathematical algorithms.

Besides working to reduce errors in the Simsys framework, it is being extended to cover more types of simulations. If we have power models for each node based on idle power and energy/operation, it is straightforward to relate power to capacity, and then extend the model to include energy. It is also possible to use additional microbenchmarks and automatic grouping of memory accesses to model each cache hierarchy level to improve HW node resolution. Finally, to further reduce errors and enhance HW resolution, HW performance counters are being introduced where appropriate in Simsys (e.g. counting cache misses).

Another enhancement to Simsys will be to provide a parameterized front end that allows generating performance quality metric results giving information as discussed in the *Q* metric examples of Section 4, but on a designer-driven basis that would allow consideration of higher level "what if" questions.

7. ACKNOWLEDGMENT

This paper is a result of a collaborative work between Intel and Exascale Computing Research Lab with support provided by CEA, Genci, Intel, and UVSQ. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the CEA, Genci, Intel, or UVSQ.

8. REFERENCES

[1] D. J. Kuck, *High Performance Computing: Challenges for Future Systems*. Oxford, UK: Oxford University Press, 1996.

[2] David J. Kuck, "Computational Capacity-Based Codesign of Computer Systems," in *High-Performance Scientific Computing*. London: Springer, 2012, pp. 45-73.

[3] William Jalby et al, "Measuring Computer Performance," in *High-Performance Scientific Computing*. London: Springer, 2012, pp. 75-95.

[4] CAPS. Codelet Finder. [Online]. http://www.caps-entreprise.com

[5] Souad Koliai et al, "A Balanced Approach to Application Performance Tuning," in *LCPC*, Newark, DE, 2009.

[6] J. C. Beyler et al, "MicroTools: Automating Program Generation and Performance Measurement," *PSTI*, 2012.

[7] William H. Press et al, *Numerical Recipes in C (2nd ed.)*. New York, USA: Cambridge University Press, 1992.

[8] D. J. Kerbyson et al, "Predictive performance and scalability modeling of a large-scale application," *ACM/IEEE conference on Supercomputing (CDROM)*, p. 37, 2001.

[9] A. Snavely et al, "A framework for performance modeling and prediction," *ACM/IEEE conference on Supercomputing*, pp. 1-17, 2002.

[10] J. McCalpin STREAM.. http://www.cs.virginia.edu/stream/

[11] Tejas S. Karkhanis et al, "A First-Order Superscalar Processor Model," *ISCA*, pp. 338-349, March 2004.

[12] Joel. Emer et al, "Asim: a performance model framework," *Computer*, vol. 35, no. 2, pp. 68 - 76, Febuary 2002.

[13] Peter S. Magnusson et al, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50-58, 2002.

[14] C.J. Hughes et al, "Rsim: simulating SMPs with ILP processors," *Computer*, vol. 35, no. 2, pp. 40-49, 2002.