Teradactyl: an Easy-to-Use Supercomputer

Augustus K. Uht

***Abstract*— Since Ada Lovelace, using supercomputers has been much harder than building them. Compilers can only occasionally automatically parallelize interesting programs. Researchers spend years developing efficient code. We propose to turn the supercomputing norm on its head: we won't use software to adapt to hardware idiosyncrasies, instead we'll make software easy with sophisticated hands-off supercomputers. We want to let scientists concentrate on science, instead of scheduling loop iterations on parallel hardware. In this paper we propose an alternative: using our Teradactyl supercomputer architecture, based on our successful Levo processor (see http://www.jilp.org , Volume 5, August). We describe and analyze Teradactyl within.**

***Index Terms*— Supercomputing, hardware, resource-flow, computer architecture.**

I. Introduction

Supercomputer programming is more torture than an art or a science. Since Ada Lovelace, using supercomputers has been much harder than building them.

The basic problem addressed herein is that supercomputer users may need months to years of programming effort to port or create their unique research applications to the latest supercomputer, greatly wasting their valuable time and decreasing the world's overall research productivity. This situation arises because of the arcaneness of all of the supercomputing programming methods used to date. These methods don't make it easy for the scientist programmer, they make it harder and force him/her to spend much time learning the nuances of computer science and computer architecture, instead of spending the time on her/his own research area. This situation is a real waste of talent, and constrains the amount of research progress in non-computer areas.

This problem is exacerbated by the frequent introduction of new computer architectures and programming models. Once an application has been ported to one machine, that machine is obsolete and another porting must take place.

As implied above, the current solutions to the programming problem are really not solutions, they merely pass the buck to the user. For example, many software libraries are available for the application programmer to use. But which library/architecture combination is best for a particular application? The application programmer needs to spend a lot of time to figure this out. And this is what we want to avoid: wasting the non-computer researcher's time.

The essence of our solution is to give the 'buck' to the computer hardware. We propose to solve the programming problems with a more robust computer architecture, one that solves the major problem of data dependency detection and minimization. Further, the proposed *Teradactyl* machine presents the classic simple sequential programming model to the application programmer, so, for example, the programmer need not try to parallelize his/her code, nor worry about scheduling calculations on the target machine, necessary today.

A. Further Motivation

In modern times, the CDC 6600 [28] and the IBM 360/91 [1] could be said to have started the modern era of supercomputing. With the addition of vector units (the Cray-1) or SIMD operation [4] compiler writers went wild trying to automatically parallelize high-level programs [15], with some success [21]. While there may have been some initial hope of parallelizing general purpose programs with complex control flow [3, 8], this was quickly dropped in favor of scientific programs with simpler control flow and, hopefully, "embarrassing" parallelism. As science and engineering of all kinds quickly progressed, it became apparent that the relevant computations required were not all that straightforward. While the control flow in such programs may be simple, the data flow can be extremely complex, especially when trying to map scheduling and communications onto a real parallel machine.

Having been out of the supercomputing field since about 1988 [32], I looked at the 2002 SC (formerly "Supercomputing Conference") proceedings to see what had happened in the ensuing 15 years. As far as I can tell, little has changed (by-the-bye, this has also been mainly true in my own main field of computer architecture); in fact, things have gotten worse. We are building bigger machines without having figured out how to program the prior ones. The useful lifetime of a supercomputer seems to be about two years, if that, after many hard years of design and construction; this is not enough time for scientists to figure out how to use the machines well; and then it all changes again. The applications are getting more demanding and more areas of science and engineering need supercomputing to make progress. (Indeed, the safety of the U.S. nuclear stockpile depends on progress in supercomputing.)

We propose to turn the supercomputing norm on its

This work was supported in part by the U.S. National Science Foundation under Grant No. MIP-9708183. Patents applied for.

Augustus K. Uht is with the University of Rhode Island, Microarchitecture Research Institute, Department of Electrical and Computer Engineering, 4 East Alumni Ave., Kingston, RI 02881 USA (telephone: +1-401-874-5431, e-mail: mailto:uht@ele.uri.edu).

head: we won't use software to adapt to hardware idiosyncrasies, instead we'll make software easy with sophisticated hands-off supercomputers. In other words, our goal is to enable the use of simple sequential programming of supercomputers, letting the hardware do the hard stuff. We want to let scientists concentrate on science, instead of scheduling loop iterations on parallel hardware.

A legitimate question to ask at this point is: won't we wind up with grossly inefficient hardware, or extremely slow programs? Maybe so, maybe not. While it is our responsibility to avoid both of these situations, certainly one can go too far in the other direction. To illustrate our claims, we argue that the best metric for supercomputer performance on a given application is not flops, not wall clock time, but rather: total *time-to-solution*. This is defined as the wall clock execution time of all runs necessary for a program, including test runs, PLUS the time spent by the user to program the supercomputer. The latter is non-trivial, including software mapping, porting, tuning, education (learning hardware and software library architectures), and thus possibly taking years, as we will see. We aim to reduce it greatly, possibly at the expense of some inefficiency or a few flops, but maybe not even those will be necessary.

Our plan is to leverage our recent successful work in high-performance uniprocessor microarchitecture [35] to design a supercomputer that self-schedules, maps, and honors dependencies with no help from the user, not even the compiler writer. A *resource-flow* model of computation will be employed, allowing rampant speculation of code while ensuring correct program execution by *time-tagging* instructions and data.

The remainder of this paper is organized as follows. Other background material is presented in Section II. In Section III the resource-flow uniprocessor called Levo is described to introduce the basic concepts. Teradactyl's architecture is presented in Section IV and its performance analyzed in Section V. We conclude in Section VI.

II. More Background

A. State of the Torture – Current Status of Supercomputing Issues

In [6] the experiences of Boeing, Intel and HP of porting the BCSLIB-EXT code from legacy code to Itanium 2 processors are related. It seems that just moving from one supercomputer architecture to another architecture is a major undertaking, even going to a machine designed for ease of parallelism extraction (by the compiler).

Programming clusters of commodity processors can be daunting [22], as can programming any supercomputer. Compounding the problem is the difficulty in predicting and modeling performance [17].

An excellent paper describing the programming of a new computational chemistry application is [5]. From this paper's Discussion section we quote the following statement, illustrating our thesis concisely:

> "Currently, the manual development and testing of a reasonably efficient parallel code for a computational model … typically *takes months to years for a computational chemist*."
> (The italics are ours.)

The authors describe efforts to build a system to reduce the time to a day or less, but this is only for one application area; of course, there are many more. Other automatic tuning systems include ATLAS [38]. Even tuning applications for uniprocessors can be difficult [37].

Even with much effort, good performance can be hard to achieve. For example, in [11] going from 1 to 128 processors on an IBM SP only gives performance gains of about 2x to 10x on seven applications. A lot of the effort goes into representing dependencies for efficient mapping and scheduling; we seek to completely avoid this.

Many libraries of system routines and other aids have been devised with great effort to ease the programmer's task. One of the best known is the Message Passing Interface (MPI) [12, 25]. But programming with such aids is still quite difficult [30].

Compiler construction has a hard time keeping up with "rapidly shifting architectures" [20]. The use of directives in program construction and compiler aids is limited, in this case for loop parallelization [9].

B. Thus: Still Need Easier Programming

Think of the sphere of influence of supercomputing if anyone could use it, especially researchers from other fields; biologists could concentrate on biology, chemists on chemistry, etc. Think of the advances computer scientists and engineers could make if their efforts could be focused other than on machine and application scheduling, mapping and tuning. There is potential here to make a large impact.

III. Introduction to Resource-Flow Computing - Levo: a Resource-Flow Uniprocessor

A. New Hardware Execution Model - Resource-Flow

Resource-flow computing [19, 31, 34, 35] defines a machine where shared resources are allocated to the temporally earliest, ready to execute, instructions, regardless of the presence or absence of correct input operands; thus, rampant speculation occurs. Levo is a realization of resource-flow computing, having active stations, time tags and distributed resources, but without centralized resources such as a register file or reorder buffer. Data dependencies are minimized and any necessary ordering is maintained with very simple hardware. Levo also employs fixed-length segmented forwarding busses that help to ensure scalability of the microarchitecture over a wide range of machine configurations or geometries. So far disjoint eager execution [36] and hardware-based full predication [35] have also been used to increase the number

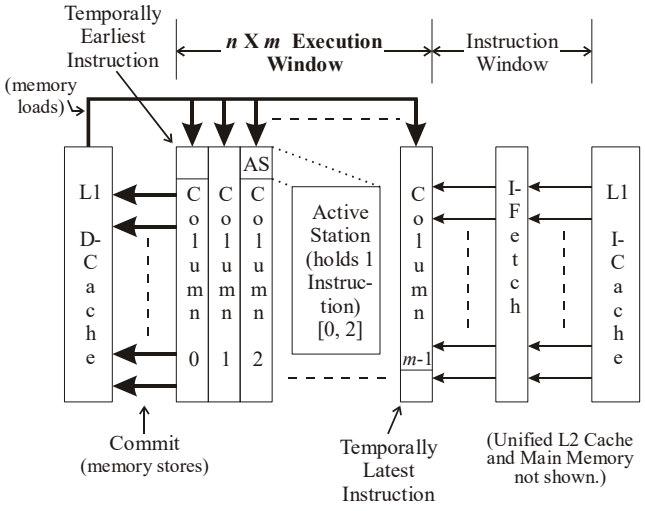
of executable instructions. These techniques expose a significant amount of instruction level parallelism and obtain high Instructions Per Cycle (IPC) ratings without increasing clock cycle time. Currently simulations of Levo executing 10 standard general-purpose (not scientific) benchmark programs from the SPECint95 and SPECint2000 suites show that Levo realizes IPC's greater than 6, on average, with realistic assumptions, and without an increase in the cycle time. Our studies also indicate that with certain architectural improvements, Levo may obtain IPC's greater than 10[14, 35].

B. Uniprocessor High ILP/IPC Realization – Levo

1) Overview

Levo consists of distributed and scalable hardware. A high-level logical block diagram of Levo is shown in Figure 1. The major novel part of Levo is the *n* X *m* instruction Execution Window (E-window).

Levo operates as follows. Instructions are fetched from the L1 I-Cache into the Instruction Window and assembled into a block one E-window column high (*n* instructions). When the first column (0) in the E-window commits, the entire E-window contents are logically shifted left and the new instruction block is shifted into the last E-window column (*m*-1). Column 0 commits when all of its instructions have finished executing: the memory store results in Column 0 are sent to the L1 D-Cache, and the Instruction Set Architecture (ISA) register results are sent to later columns. Processing resources are located uniformly throughout the E-window. All instructions in the E-window, including memory operations, are eligible for execution at any time. Store results, as well as register operation and branch operation results (predicates), are broadcast forward (to the right) in the E-window and snarfed by instructions with matching operand addresses. Load requests are satisfied either from earlier in the E-window or directly from the L1 D-Cache.



Processing Elements (PEs) are distributed among AS's.

Figure 1. Levo high-level logical block diagram. The Execution Window is the key element.

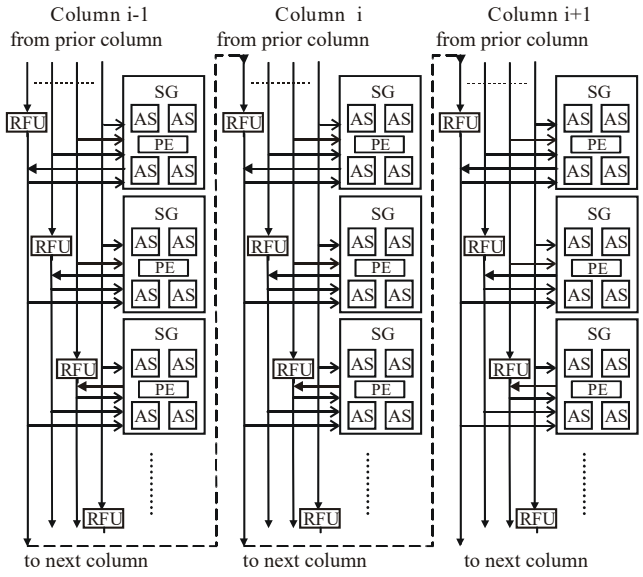


Figure 2. Generic Levo E-window details. A segmented (spanning) bus's length does not change as columns are added to the machine. (Physically, the last column connects to the first column, forming a loop of columns. In a typical floorplan the loop is constructed so that the delay across all bus segments is the same and the delay from column to column is nil.) In this example the spanning bus length is 3 SG's. Each RFU also snoops the other buses at its level in the same column (not shown), to maintain RFU consistency for its SG.

There are two key novel features of the E-window that make it scale and ensure that each operand (eventually) gets the right result as its input. First, the broadcast bus is divided into *segments*, each one typically a column long. The bottom, or end, of one segment is coupled to the top of the next segment via storage elements having a small delay. Thus, additional columns can be added to the E-window without impacting Levo's cycle time.

The second novel feature is Levo's use of *time tags*. Each instruction in the E-window has a unique time tag corresponding to its position in the E-window. The time tags provide the proper result-operand linkage with scalable hardware, since all comparisons are made simultaneously with an amount of hardware linearly proportional to the machine size. The time tags are used for all dependency checking and all data: memory, register and predicate.

In detail, the E-window holds *nm* Active Stations (AS). An Active Station is a more intelligent form of Tomasulo's reservation station [29]. Each AS holds one instruction. Small numbers of physically close AS's form Sharing Groups (SG); see Figure 2. Processing Elements (PE) are assigned to each sharing group, typically one PE per SG. Each AS in the E-window has a corresponding integer *time tag* indicating its instruction's nominal temporal execution order. Time tags are formed by the concatenation of the AS's E-window column number and row number. Every time column 0 commits, the column part of all of the time tags in the machine is decremented by one; hence, time tags are easily recycled and never need to be more than several bits long.

Levo's microarchitecture is alterable to match any ISA. So far we have fully realized one GP ISA, the MIPS-1, in our simulator and obtained high performance. No compiler modifications are necessary for Levo. Therefore both legacy and standard sequential code can be executed.

2) Basic Levo Time Tag and Active Station Operation

Levo uses novel time-tagged active stations to realize rampant speculative execution of code. No explicit renaming registers or reorder buffer are used.

In Levo the time tags are constructed and used in novel ways: to both enforce programmatic execution (no Program Counter is used) and minimize dependencies; e.g., this is used to eliminate WAR (Write After Read) dependencies in both register and memory accesses.

The basic operation of time-tagged instructions is shown in Figure 3. Time-tagging assumes the broadcast of instruction result information on a single logical bus in one direction, snooped by all later active stations. The basic conditions necessary for an active station to snarf or read the contents of the bus and allow the active station's instruction to fire are:

1. The broadcast time tag on the bus must be less than the time tag of the active station.
2. The broadcast time tag must be greater than or equal to the time tag of the snarf register (last snarfed time tag: LSTT).
3. The value of the broadcast datum must be different from the value of the existing operand in the active station (if any).

If these conditions are met, the following happens:

1. The broadcast time tag is copied into LSTT.
2. The datum is copied into the active station's corresponding operand's value register.
3. The active station's instruction is sent to its corresponding PE with its operands for execution.

Once the instruction is executed:

1. The execution result is broadcast on the bus, contention permitting, with its register address and the active station's time tag.
2. The active station disables itself for further execution until another datum is snarfed.
3. The entire process repeats.

Thus, only the closest result (with an earlier time tag value) present on the forwarding bus is used as the final input operand value for an instruction; this both enforces correct program execution and minimizes data dependencies. The inefficiency of Case 1 in the figure, I9 firing twice, does not substantially adversely affect performance.

Since each time tag corresponds to exactly one instruction, and hence exactly one Active Station, time-tagging has linear cost growth, O(k), with the number of instructions held in the execution window. Therefore, it is scalable. Further, its execution algorithm is simple.

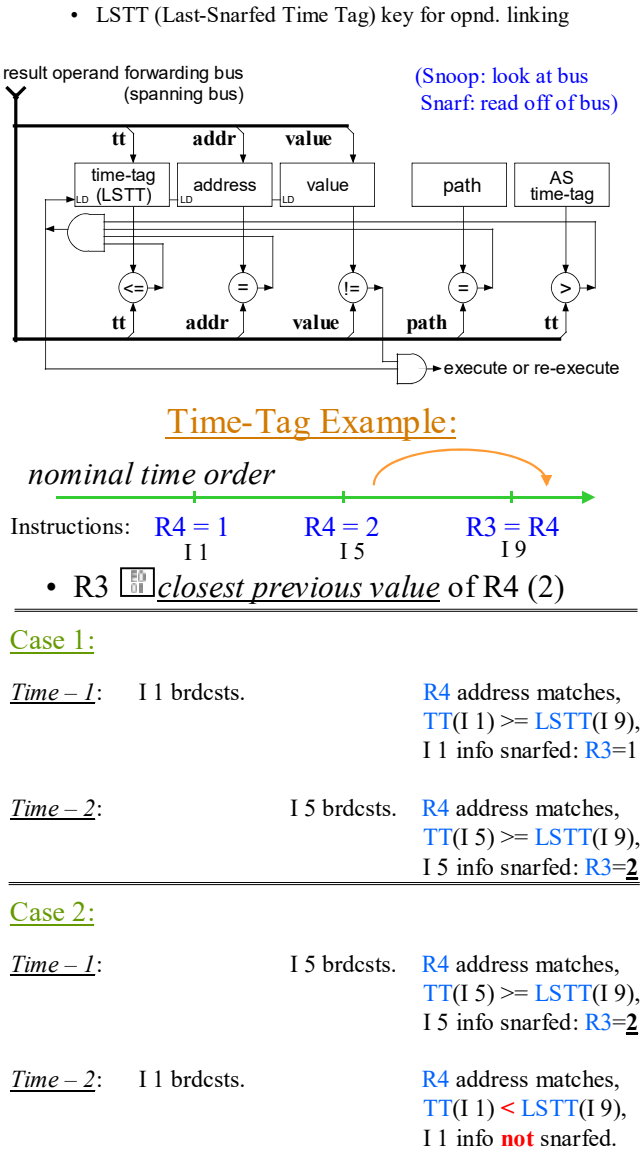


Figure 3 Active Station logic and Time Tag example. In the example, in Case 1, I1 executes before I5, while in Case 2, I1 executes after I5. In both cases I9 ends up with the correct datum, '2.'

With memory operands, the memory address is used in place of the register address.

Time tags are used in common processors to squash instruction results occurring after a mispredicted branch, as well as to maintain instruction order in general. A timestamping method was originally proposed for microarchitectures in the Warp Engine [7], which in turn was based on a novel simulation technique [13]. In the latter, a strict ordering of timestamps is maintained at receiving processes (in our case, instructions); Levo does not require this. The Warp Engine relied on the use of either floating point numbers or very large integers for the time tags, only tagged memory references, required the use of bandwidth-consuming "anti-messages" for mis-speculation rollback, and used the tags only for control-flow ordering.

3) Segmented Result Buses ⇒ Scalable Microarchitecture

In Levo, segmented or spanning buses are used to propagate active station results to later active stations. This avoids a performance penalty because an instruction's result is likely to be used soon after it has been created [2, 10, 18, 26]. Adjacent segments are connected via Register Forwarding Units (RFU), which introduce a small delay, usually one cycle, from segment to segment; see Figure 2. The idea is that the later in the E-window a result is used, the more likely it is to be used later in time, and the delays introduced by the RFU's will be hidden. Segment length is independent of column height. Since the length of segments need not change with the size of the machine, the spanning buses help make Levo scalable.

RFU's hold versions of the ISA register state. Time tags are forwarded along with their corresponding register values by the RFU's onto later bus segments. RFU's also provide a filtering function: multiple writes to the same ISA register in an RFU are combined, keeping the later time tag and value, and only one result value for that register is forwarded.

There is one RFU per sharing group and nominally one spanning bus per RFU. There are also Memory Forwarding Units (MFU), Predicate Forwarding Units (PFU), and spanning buses (not shown) for each of the corresponding data. The number of ports to/from RFUs, MFUs and PFUs are small and are constant with respect to the size of the machine; this also helps ensure scalability.

Other novel features are: the elimination of a centralized register file, and the simplification of state commitment, both by using RFUs. To see this, assume in Figure 2 that column *i*-1 is column 0, where instructions are committed. By the time an RFU's state reaches column 0, it contains the equivalent of what would normally be thought of as the ISA register state. Since the register values have already been broadcast to RFU's in later columns, and since a new column's (*m*-1) RFU's are initialized with the contents of the prior column's RFU's, there is always at least one RFU in the E-window that holds the equivalent of the ISA state, no matter the time difference between writing and reading an architectural register; therefore it is unnecessary to save the ISA register state in a separate register file. The same is true of the predicate state. The memory values, however, must be written to the L1 D-cache, since an MFU cannot hold all possible memory locations.

Sometimes instructions must request operands from earlier in the E-window. This is done via *backwarding buses* (not shown), following the same paths as the forwarding buses, just going in the opposite direction. An example of this is a Load instruction that has computed its memory address and is ready for execution, but has not yet snarfed a corresponding memory datum. The Load sends its request backwards in the E-window, along a backwarding bus; the request contains the memory address of the requested datum. A Store with a matching address will snarf and stop the backwarding request, and broadcast its Store value on a forwarding bus, in the usual way. The Load will then snarf the datum. Should there be no matching Store, the request is satisfied from the memory hierarchy; the latter is accessed simultaneously with the E-window search, just in case there is no matching store earlier in the E-window.

C. Initial Levo Experimental Results

We have performed extensive cycle-accurate simulations on the Levo model [18, 19, 31, 35]. Briefly, 10 benchmarks from the SPECint95 and SPECint2000 suites were compiled into MIPS-1 code and executed on the simulator. These benchmarks are general purpose and typically have complex control flow, e.g., gcc; they are notoriously hard to execute well. While the IPC realized for a reasonable configuration and a main memory latency of 100 cycles is only about 5 IPC, we have discovered that if instruction fetch is improved the harmonic mean IPC goes up to 7-to-17, depending on the configuration. We are currently both addressing this issue as well as implementing other IPC enhancing techniques.

We have also found that main memory latency is tolerated well. Increasing the latency from 25 to 800 cycles reduces the performance by only 6%.

Thus, the resource flow model works well on really tough code. Usually the SPECfp codes execute much better on high-IPC machines than the SPECint's do. Since the SPECfp codes are much more similar (if not the same) as those executed on supercomputers, the implication is that a supercomputer using the resource flow model might execute its applications extremely well and with little if any additional programming effort.

D. Why Levo is Relevant to Supercomputing

The key problem with current supercomputing is the detection and minimization of data dependencies. Levo handles data dependencies routinely with scalable hardware and little effort. Resource-flow execution takes advantage of rampant speculation, which can be used as a tool to effectively schedule and map computation on supercomputers automatically, with no programmer involvement. Time tags make all of this possible. It is very easy to add resources with the Levo model; this is also true in our proposed supercomputer. Programming is made simple, once again.

IV. TERADACTYL – RESOURCE-FLOW SUPERCOMPUTER

A. Architecture and Overall Operation

A block diagram of the overall Teradactyl logical architecture is shown in Figure 4, for a 16-processor implementation. It is intended that Teradactyl scale to 100's of 1,000's of processors. The simple logical "wagon-wheel" structure shown is realized in physical hardware with a more linear and scalable construction; more on this in Section *4)*.

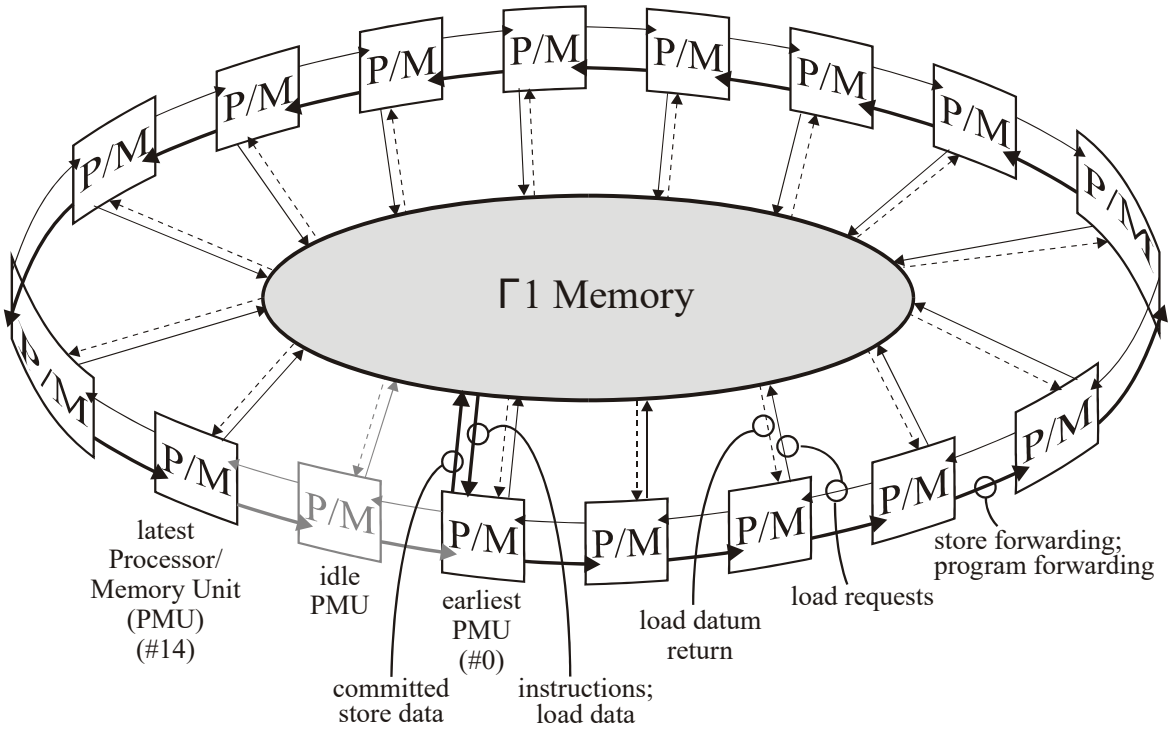


Figure 4. Teradactyl block diagram (logical). Paths to or from Γ1(Gamma1) memory and PMU's are called "gamma rays." Gamma rays for 'committed store data' and 'instructions' are shown only for the earliest PMU (#0); such physical connections are also needed for the other PMU's, but only those for PMU #0 are active at any given time. Gamma rays for 'load datum return' and 'load request' are active for all PMU's all the time. All gamma ray bandwidth requirements are small.

Roughly speaking, a complete Teradactyl corresponds to a complete Levo machine. In particular, a Teradactyl PMU (Processor/Memory Unit) corresponds to a Levo column. Internally, a PMU is similar in construction to a Levo machine. Hence, Teradactyl is essentially a macro-Levo constructed of smaller micro-Levo's.

1) Basic Structural Description

The central part of Teradactyl is the Γ (gamma) 1 memory, so called because it is above the main memory (Γ0) in the memory hierarchy (gamma=upside-down 'L'; a regular 'L' is used below the main memory). Γ1 holds not only the coherent data memory of the entire computer, be it semiconductor, disk, tape or a combination of them, but also the program or instruction memory, and the overall control unit of Teradactyl.

All of the application's execution occurs in the ring of Processor/Memory Units (PMU's) around Γ1. Each PMU executes a large part of the overall computation, much more than is held in each PMU's (~Levo) E-window. The PMU's and the Γ1 memory communicate via "gamma rays," or spokes of the wheel. The load-request and return gamma rays to and from every PMU are relatively low-bandwidth connections.

In the desired plan, there will be physical gamma rays for instructions and committed store data for all of the PMU's, but only one PMU's such gamma rays will be active at any given time, and usually for a relatively long time (micro- or milli-seconds). Thus, the actual bandwidth required at any given time to and from Γ1 should be relatively small.

Teradactyl's size is easily increased by inserting more PMU's in the loop.

2) General Operation

Initially, none of the PMU's contain any data or instructions or are doing any computation. To start the execution of an application, G1 sends instructions (the application machine code itself, a standard non-parallelized imperative program) to an arbitrary PMU, which is then labeled with time tag (TT) #0. Thus, the ring of PMU's corresponds to Levo's E-window, and an individual PMU corresponds to an E-window column. The overall time tag of an instruction is thus its PMU number concatenated with its E-window time tag.

PMU #0 allocates for itself an estimated useful amount of work (a '*chunk*'), then passes on future computation (instructions) to the next PMU (#1), PMU #1 gets its estimated share and passes the buck to PMU #2, and so on counter-clockwise around Γ1; see Figure 4. PMU's continue to be added to the computation until PMU #0 is reached from the other side; the hardware is then fully allocated.

All of the PMU's execute their own share of the computation simultaneously. Resource-Flow rampant speculative execution is used. Thus, for example, PMU #4 could be computing at the same time as PMU #1, even though by the nominal sequential order PMU #1's computation should all be performed first.

What makes sure we get the right answer? Similarly to Levo, stores are effectively forwarded from PMU to later PMU and so on counterclockwise around the ring. The forwarding is adjusted so that the net bandwidth is linearly proportional to the number of PMU's. Any PMU receiving a store with an address matching a valid datum in its Γ0 memory may have to redo part or all of its computation, since an input may have changed. Since most scientific computation is data independent, we believe this will be a rare circumstance, and instead Teradactyl will realize close to the full parallelism inherent in the application.

As a simple example, consider the addition of two matrices of vast sizes. The code goes to PMU #0, it takes, say, a million or a billion elements for itself, then PMU #1 takes the next million elements, and so on. As soon as a PMU has its code loaded, it begins execution. It will start to generate load requests, which are sent back to Γ1 via the backwarding ring going clockwise in Teradactyl or via the PMU's own gamma rays. As Γ1 receives the data requests, it satisfies them by sending the data out to the ring, and hence the requesting PMU.

For a more complex example, consider another matrix operation that occurs after the matrix addition of the last paragraph. Assume the original matrices used as the inputs to the addition are sparse (their elements are mainly equal to 0).

In the Teradactyl model, the second matrix operation will commence independently of the first operation, possibly even before it, even the second operation's code is later in the Teradactyl ring; this is due to the rampant speculation of the Teradactyl model. The second operation will use whatever it predicts are the values of the results of the first operation; since the first operation's output matrix is sparse, most of the time the second operation will use the correct values for its inputs, that is, it will have correctly predicted the values of its inputs.

Now, let's say the first operation produces an element of its result matrix that is non-0. Like the other results, this value is forwarded around the Teradactyl ring with its address. If a sub-operation in the second matrix operation has a matching address, and its predicted input value differs from the broadcast actual value (unlike in the previous 0 broadcasts), the sub-operation will recompute its result, which in turn will be broadcast around the ring, updating later speculative computations as needed, and so on.

Therefore, matrix operations are executed in parallel, with data dependent operations eventually sequentialized as necessary, and only when necessary.

The communication around the ring will be very fast (nanoseconds from PMU to adjacent PMU), especially as compared to the execution time of each chunk in a PMU. In essence the entire computation is pipelined around the ring.

When PMU #0 finishes executing its chunk, it sends its computed data to Γ1 for permanent storage. PMU #0 is then freed, and may be reallocated to the other end of the ring. Also, all of the remaining allocated PMU's must decrement their PMU time tag values, in order to recycle them.

Therefore the ring holds a rotating window of computation, advancing counter-clockwise as the computation progresses.

3) Related Work

Clearly the big picture of Teradactyl bears resemblance to architectures past. The Denelcor HEP [23] had a rotating window of threads' state, each thread being executed a part at a time (until it blocked, say on a memory reference). However, there was effectively only one processor, so while efficiency was high, realized performance was less so.

Perhaps the closest machine to Teradactyl is the *systolic array* class of machines, originally formulated by Kung and Leiserson [16]. There are a number of differences between the two machines. First, in Teradactyl output data (results) do not flow through the array to their final destination (memory), rather the results stay with the processor where they were computed, and are saved directly from there. Secondly, and perhaps most importantly, systolic arrays were heavily tied to the scheduling and mapping capabilities of their compiler, exactly what we are trying to avoid. Thirdly, the overall structure of a systolic array could be one dimensional, two dimensional, or something else, but always trying to match the targeted computations. Teradactyl targets all scientific large-scale computations, regardless of problem, program or data structure, and without the compiler's assistance.

There is also a resemblance to *counterflow* processors [27], but the latter were asynchronous machines aimed at the microprocessor/single-chip level, with instructions going in one direction and interacting with data going in the opposite direction. Teradactyl is dissimilar on all accounts.

The *Decoupled Access/Execute* computer [24] is another type of machine separating flows of data or instructions. It was also targeted for uniprocessor construction, nor did it make use of time-tagging, to our knowledge.

4) Physical Realities and Possible Solutions

As shown in Figure 4, a logical Teradactyl does not scale: the total volume is proportional to the radius squared. However, there are numerous possible physical solutions, especially given the linear/ring nature of the machine.

First, it is possible that many PMU's would fit on a single chip, or at least many CPU's. This would immediately but not dramatically increase the density of the machine.

More likely would be a folding of the ring, again around a common core, like a cylindrical accordion. The ring could also be "wound" around the common core on a cylinder, spiraling up the outside of the cylinder then connecting to the inner PMU's, spiraling down the inside of the cylinder and then reconnecting to the first PMU at the bottom. There are many physical topologies that would support the scaling of Teradactyl.

The reliability and fault-tolerance of 1,000 processor

machines must always be considered. Teradactyl makes fault-recovery relatively easily. When a PMU fails, it is taken out of the wheel, and the resulting free ends of the wheel are logically connected together. In the worst case, all computation after the faulty PMU is squashed, and is repeated with the working PMU right after the faulty PMU taking the faulty PMU's place and re-executing its code. The programmer is in no way involved in this process. Even the system operator need do little, other than record the location of the faulty PMU for later replacement.

Teradactyl's memory system architecture must still be honed to meet the needs of its likely applications. Also, the code chunking algorithm must be tuned and fleshed out. We are currently working on these issues.

B. Processor/Memory Unit (PMU) Microarchitecture

While it would be economically desirable to use a commodity processor such as an Intel or an AMD chip for the PMU, Teradactyl's performance would likely suffer. This is because there is no externally-accessible indication of an instruction's sink's time tag, or the equivalent. Thus, a datum or other mispeculation would necessitate the squashing of all later instructions in all later PMU's in the ring. This is unacceptable. Therefore we will use a Levo-based microarchitecture for the PMU.

A complete PMU in Teradactyl is analogous to an E-window column in Levo. The PMU's microarchitecture is similar to Levo's; limited space precludes its precise description here.

We are currently developing a Levo prototype. We have already obtained some of its physical hardware components. Our experience with this prototype will help us with the detailed architecture of a PMU. In fact, we may be able to realize one or more PMU's with the Levo prototype, since the prototype will be constructed out of programmable logic. Therefore, we may be able to realize a very small version of a Teradactyl in hardware and test its operation.

We hope to have the Levo prototype constructed and tested by early 2005. We do not yet know if a chip realization will be attempted. A lot depends on how successful the prototype is, and if it is picked up by industry.

V. Teradactyl Analysis – Achieving Peta-Ops

We are interested in two estimates: how many processors are needed to execute both a sustained Tera-op, as well as a sustained Peta-op? (Note: we are not interested in peak performance, but rather the *sustained* performance which is exhibited during actual code execution.) Further, can these machines be built?

We will take a Levo PMU as a starting point, using existing simulation results and hardware estimates [14, 35]. We assume Levo will operate at 5 GHz, a conservative estimate for a high-end processor a few years from now. We further assume that Levo will realize 10 IPC on average on scientific codes, a reasonable if not conservative estimate. (This may actually be achievable today. We have not simulated the SPECfp codes, which should do better than the SPECint codes; the latter give IPC's of about 6, today.) Note that this all includes high-latency memory accesses and other realistic assumptions.

Therefore each PMU will yield a sustained 50 Giga-ops of performance. 20 PMU's arranged in a Teradactyl wheel will thus give a sustained Tera-op. Let us assume that not all PMU's will be efficiently utilized, and call it 25 PMU's for a Tera-op. Then a Peta-op will take 25,000 PMU's, and dissipate around 1-2 Megawatts. The latter is exhibited by today's supercomputers, and it may decrease as a result of all of the low-power processor research underway today. We call this size of Teradactyl a *Petadactyl*.

By way of further comparison, the fastest supercomputer today, the Japanese Earth Simulator, has a peak performance of about 35 Tera-ops. Much of this is realized in sustained program execution of at least some programs. Other supercomputers have peak performances in the 5-7 Tera-ops range, and have difficulty realizing a Tera-op in sustained performance on many codes.

While Petadactyl's 25,000 processors look daunting, remember that the Teradactyl programming model is extremely simple; no programmer-based parallelization or resource allocation is performed. No communications strategies need be devised. Further, greater advances in IPC realization are likely before such a machine is built, and processor clock-rates may be greater [33], leading to a reduction in the total number of PMU's needed.

VI. Conclusions

Teradactyl is a new supercomputer architecture that greatly reduces the time needed for programmers to make use of high-end computers. In particular, non-computer specialists will find it easy to program Teradactyl and obtain solid performance, without the machinations currently required of such researchers.

Teradactyl is scalable by design, can easily reach Tera-ops speeds. Peta-ops performance is readily obtained when the requisite larger number of processors is used, giving a: "Petadactyl." Such a change will require little or no porting of program code by the end users. If anything, the codes will be simplified, as compared with the code versions used on traditional supercomputers.

Clearly, there is much to be done before we can make a strong experimental case for Teradactyl. The current Levo simulator, FastLevo, must be adapted for the particular microarchitecture needed for Teradactyl PMU's, and a Teradactyl simulator must be written to verify its high performance.

In terms of prototypes, the goal is to first build a Levo processor (its design is underway), then a 25 processor Teradactyl, then a 25,000 processor Petadactyl.

While the challenges in realizing our goals are great, we are confident of success. In order to solve hard problems, especially old ones like automatic extraction of parallelism,

one must start from a completely different perspective, and be willing to rethink all of the assumptions that have hampered success. We are doing this.

REFERENCES

[1] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," IBM Journal of Research and Development, vol. 11, no. 1, pp. 8-24, January 1967.

[2] T. M. Austin and G. S. Sohi, "Dynamic Dependency Analysis of Ordinary Programs," in Proceedings of the 19th Annual International Symposium on Computer Architecture. Gold Coast, Australia: IEEE and ACM, May 1992, pp. 342-351.

[3] U. Banerjee and D. Gajski, "Fast Execution of Loops With IF Statements," IEEE Transactions on Computers, vol. C-33, no. 11, pp. 1030-1033, November 1984.

[4] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV Computer," IEEE Transactions on Computers, vol. C-17, pp. 746-757, August 1968.

[5] G. Baumgartner, D. E. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C.-C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan, "A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry," in Proceedings of SC2002: High Performance Networking and Computing. Baltimore, Maryland, USA, November 16-22, 2002.

[6] J. T. Betts, J. G. Lewis, and D. K. Wah, "Effect of Tuning Legacy Applications for Contemporary Processors and Contemporary Memory Hierarchies: A Sparse Matrix Case Study," in Proceedings of SC2002: High Performance Networking and Computing. Baltimore, Maryland, USA, November 16-22, 2002. Masterworks Session, Computer Aided Engineering II, URL: http://www.sc-conference.org/sc2002/.

[7] J. G. Cleary, M. W. Pearson, and H. Kinawi, "The Architecture of an Optimistic CPU: The Warp Engine," in Proceedings of the Hawaii International Conference on Systems Science (HICSS), vol. 1: University of Hawaii, January 1995, pp. 163-172.

[8] R. G. Cytron, "Doacross: Beyond Vectorization for Multiprocessors (Extended Abstract)," in Proceedings of the 1986 International Conference on Parallel Processing, August 1986, pp. 836-844.

[9] S. Dong and G. Karniadakis, "Dual-Level Parallelism for Deterministic and Stochastic CFD Problems," in Proceedings of SC2002: High Performance Networking and Computing. Baltimore, Maryland, USA, November 16-22, 2002.

[10] M. Franklin and G. S. Sohi, "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors," in Proceedings of the Twenty-Fifth International Symposium on Microarchitecture (MICRO-25): IEEE and ACM, December 1992, pp. 236-245.

[11] L. Grigori and X. S. Li, "A New Scheduling Algorithm For Parallel Sparse LU Factorization with Static Pivoting," in Proceedings of SC2002: High Performance Networking and Computing. Baltimore, Maryland, USA, November 16-22, 2002.

[12] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, MPI - The Complete Reference, vol. 2, The MPI Extensions: MIT Press, 1998.

[13] D. Jefferson, "Virtual Time," Transactions on Programming Languages and Systems, vol. 7, no. 3, pp. 404-425, July 1985.

[14] A. Khalafi, "Exploring Multipath Execution on a Distributed Microarchitecture," Ph.D. thesis, Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA, 2003.

[15] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," in Proceedings of the Fourth International Computer Software and Applications Conference, October 1980.

[16] H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," in Proceedings of the SIAM Sparse Matrix Symposium, Duff and Stewart, Ed. Philadelphia, PA, USA: SIAM, 1978, pp. 256-282.

[17] C. Lu and D. A. Reed, "Compact Application Signatures for Parallel and Distributed Scientific Codes," in Proceedings of SC2002: High Performance Networking and Computing. Baltimore, Maryland, USA, November 16-22, 2002.

[18] D. Morano, A. Khalafi, D. R. Kaeli, and A. K. Uht, "Implications of Register and Memory Temporal Locality for Distributed Microarchitectures," Dept. of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA, Technical Report, October 2002, URL: http://www.ece.neu.edu/groups/nucar/publications/intervals.pdf.

[19] D. Morano, A. Khalafi, D. R. Kaeli, and A. K. Uht, "Realizing High IPC Through a Scalable Memory-Latency Tolerant Multipath Microarchitecture," in Proceedings of the Workshop On Chip Multiprocessors: Processor Architecture and Memory Hierarchy Related Issues (MEDEA2002), at PACT 2002. Charlottesville, Virginia, USA, September 22, 2002. Also appears in ACM SIGARCH Computer Architecture Newsletter, March 2003, URL: http://www.ele.uri.edu/~uht/papers/MEDEA2002final.pdf.

[20] D. Parello, O. Temam, and J.-M. Verdun, "On Increasing Architecture Awareness in Program Optimizations to Bridge the Gap between Peak and Sustained Processor Performance -- Matrix-Multiply Revisited," in Proceedings of SC2002: High Performance Networking and Computing. Baltimore, Maryland, USA, November 16-22, 2002.

[21] C. D. Polychronopoulos and U. Banerjee, "Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds," IEEE Transactions on Computers, April 1987. Special Issue on Parallel and Distributed Processing.

[22] C. Seitz, "The Architecture, Programming, and Applications of Clusters," in Proceedings of SC2002: High Performance Networking and Computing. Baltimore, Maryland, USA, November 16-22, 2002. Masterworks Session, Infrastructure III, URL: http://www.sc-conference.org/sc2002/.

[23] B. J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer," Society of Photo-optical Instrumentation Engineers, no. 298, pp. 241-248, 1981.

[24] J. E. Smith, "Decoupled Access/Execute Computer Architectures," in Proceedings of the 9th Annual Symposium on Computer Architecture: IEEE and ACM, April 1982, pp. 112-119.

[25] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, MPI - The Complete Reference, vol. 1, The MPI Core, second ed: MIT Press, 1998.

[26] G. S. Sohi, S. Breach, and T. N. Vijaykumar, "Multiscalar Processors," in Proceedings of the 22nd Annual International Symposium on Computer Architecture: IEEE and ACM, June 1995.

[27] R. F. Sproull, I. E. Sutherland, and C. E. Molnar, "The Counterflow Pipeline Processor Architecture," IEEE Design & Test of Computers, vol. 11, no. 3, pp. 48-59, 1994.

[28] J. E. Thornton, "Parallel Operation in the Control Data 6600," in Proceedings of the Fall Joint Computer Conference, 1964, pp. 33-40.

[29] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of Research and Development, vol. 11, no. 1, pp. 25-33, January 1967.

[30] J. L. Traff, "Implementing the MPI Process Topology Mechanism," in Proceedings of SC2002: High Performance Networking and Computing. Baltimore, Maryland, USA, November 16-22, 2002.

[31] A. Uht, A. Khalafi, D. Morano, M. d. Alba, and D. Kaeli, "Realizing High IPC Using Time-Tagged Resource Flow Computing," in Proceedings of the Euro-Par 2002 Conference, Springer-Verlag Lecture Notes in Computer Science. Paderborn, Germany: ACM, IFIP, August 28, 2002, pp. 490-499. URL: http://www.ele.uri.edu/~uht/papers/Euro-Par2002.ps.

[32] A. K. Uht, "Requirements for Optimal Execution of Loops with Tests," in Proceedings of the International Conference on Supercomputing. St. Malo, France, July 4-8, 1988, pp. 230-237.

[33] A. K. Uht, "Uniprocessor Performance Enhancement Through Adaptive Clock Frequency Control," in Proceedings of the SSGRR-2003w International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, e-Medicine, and Mobile Technologies on the Internet. L'Aquila, Italy: Telecom Italia, January 6-12, 2003. Invited and refereed paper, URL: http://www.ele.uri.edu/~uht/papers/SSGRR2003wFnlUht.pdf.

[34] A. K. Uht, A. Khalafi, D. Morano, T. Wenisch, M. de Alba, and D. Kaeli, "Levo: IPC in the 10's via Resource Flow Computing," IEEE TCCA Newsletter, Special Issue, December 2001. Presented at PACT 2001 Work-In-Progress (WIP) Session, September 2001., URL: http://www.ele.uri.edu/~uht/papers/LevoWIPPACT01.pdf.

[35] A. K. Uht, D. Morano, A. Khalafi, and D. R. Kaeli, "Levo - A Scalable Processor With High IPC," The Journal of Instruction-Level Parallelism, vol. 5, August 2003. URL: http://www.jilp.org/vol5.
[36] A. K. Uht and V. Sindagi, "Disjoint Eager Execution: An Optimal Form of Speculative Execution," in Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28). Ann Arbor, MI, November/December 1995, pp. 313-325. URL: ftp://ele.uri.edu/pub/uht/micro95.ps.
[37] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply," in Proceedings of SC2002: High Performance Networking and Computing. Baltimore, Maryland, USA, November 16-22, 2002.
[38] R. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software (ATLAS)," in Proceedings of Supercomputing '98, 1998.