The Design of a Fault-Tolerant, Real-Time, Multi-Core Computer System

Kim P. Gostelow
Jet Propulsion Lab, California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
818-354-4185
kim.p.gostelow@jpl.nasa.gov

Abstract—The goal is a fault-tolerant, self-aware, low-power, multi-core computer for space missions with thousands of simple cores, achieving speed through concurrency. A second goal is that the system is not difficult to program. The proposed machine decides how to achieve concurrency, in real-time, rather than programmers who now spend considerable effort carefully orchestrating every data item's location and movement. Closely related, fault-tolerant and power-aware re-organizing behavior is automatic. The driving features of the system are: simple hardware that is modular in the extreme, with no shared memory, and software with significant run-time re-organizing capability.[1] [2]

TABLE OF CONTENTS

1. INTRODUCTION

The era of multiprocessor/multi-core computing is here. Intel[2], IBM[4], Tilera[7], and others have built chips with dozens of cores. While some are viewed as serious contenders for spacecraft application, this paper suggests that simpler is even better. Spacecraft are a special application, and this paper describes a multi-core system catering to spacecraft needs. In particular, fault tolerant, real-time operation, and power management, are two areas that multi-core can address directly. Key to both fault-tolerance and power management is moving programs and data without interrupting the system.

2. SYSTEM OVERVIEW

Associated with each core is the core's memory. A core and its memory are called a *processor*, and there are a few hundred processors per chip (please see Figure 1 below, though the figure shows only a few processors per chip). There is no shared memory. Instead, a processor connects to its neighbors through a high-speed data link. Messages are sent to a neighbor switch, which in turn forwards that message on to its neighbor until reaching the intended destination. Except for the neighbor connections, processors are isolated and independent of each other.

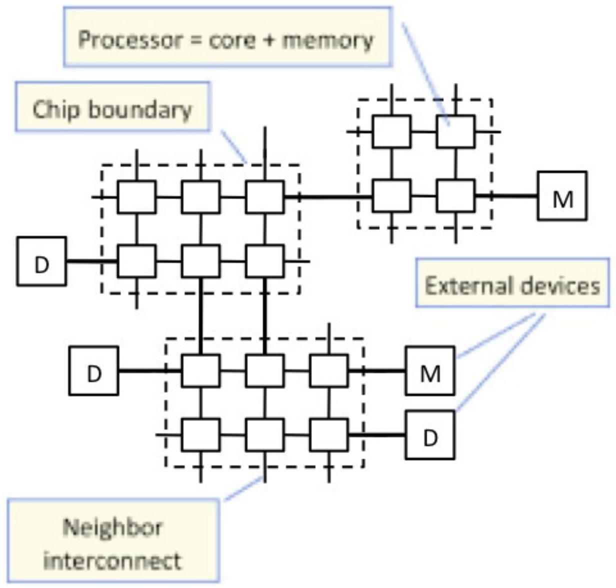


Figure 1 – Hundreds of interconnected processors with no shared memory

Some of the processors at the periphery of each chip are a little different than the inner processors, as they are able to drive off-chip to external devices or memory processors. The processors on the periphery also connect chip-to-chip, thus building up a large processor net. There is no particular topology to the larger net, as a function at each

[1] 978-1-4244-7351-9/11/$26.00 ©2011 IEEE.
[2] IEEEAC paper #1607, Version 2, Updated October 26, 2010

processor allows it to forward a message in the correct direction. Some chip-to-chip connections are not necessarily nearest neighbor, providing short cuts for some of the longer physical distances. The peripheral processors also provide the connections to sensors, actuators, radios, science instruments, and all the other devices with which the computer system interacts.

A requirement on this system is very low power. The design sacrifices speed per processor, but gains silicon to include more processors per chip, faster interconnect, and yield. Rejection of shared memory is both a programming and a hardware design point. Previous studies have shown the power cost of global memory and caches to be high[17]. Non-shared memory will increase communication needs, but instead of investing silicon and power in a global memory system, we build a faster interconnect, where speed increases can be used by all computations, not just those that access global memory. Chip stacking is especially easy because each processor is independent; all wires on a given chip are within a single processor, or part of a neighbor-to-neighbor connection. This point of view shares much with the motivation of the Execube [4] system, though the goals here are different. The emphasis here is on fault-tolerance and the ability to move computations in real-time from one part of the machine to another. Ongoing computations may move due to

(1) Processor failure

(2) Locality considerations that make it more efficient to perform the computation in a different part of the machine

(3) Some portion of the machine is to be powered off to reduce power use

(4) Some part of the system has been physically damaged.

This movement of data and functions from one area to another is fundamental. It requires that the machine and its computations are self-aware, and that it consider where the work is done, at what rate, and how much power to expend. Closely related to moving computations is creating copies of computations, which is at the core of some forms of computational redundancy: repeating an instance of a given computation, as in TMR (Triple-Modular Redundancy), temporal redundancy, or check-pointing, where either an exact copy or a different version of the same computation is used.

3. The Model of Execution

Execution is based on a functional paradigm rather than von Neumann. In the von Neumann model

(1) The instructions manipulate the contents of storage locations in a memory.

(2) The machine executes instructions sequentially, that is, the operation executes when the program counter reaches it.

The functional model is based upon

(1) A function executes as soon as the necessary data arrive.

(2) Variables are mathematical variables – they have a value – they are not memory cells; for example, no variable's value is computed more than once.

The term dataflow [8, 9] has also been used to describe the functional model, as it pictures input data values flowing through the functions that produce new output data values. The functional model of execution is analogous to asynchronous digital circuit operation, while the von Neumann model is similar to clocked circuits.

4. Concurrency In Functional Programs

Since some readers will be unfamiliar with the advantages of functional programs in multi-core platforms, we discuss this point here. We show some example programs that progress from a non-functional program to pure functions, and we discuss the compiled version of each and how the proposed machine can execute the resulting code.

In functional programs, any two expressions may execute concurrently, and any two functions not dependent on one another may also run in parallel, where a function f is dependent on function g if and only if an input argument of f is (directly or indirectly) an output result of g. Functional programs thus exhibit implicit concurrency. There is no need for explicit message passing or other concurrency-control statements in the program. Although functional programs can be written in non-functional languages, functional languages such as Haskell [11], and especially Fortress [6], even cater to multi-core implementations, for example, by internally using the parallelism inherent in trees rather than linear lists (see below).

Writing functional code to achieve concurrency is in contrast to other approaches. Investigators have looked for many years at how to make sequential programs run in parallel [16]. There are many reasons for doing so: the large number of expensive sequential programs already debugged and running; or, the view that the task of programming concurrent systems is too hard for people to do, so let a few experts work it out while the rest of us write the code sequentially and they'll just "make it work". The more popular programming paradigms for concurrent machines such as MPI [12] can be hard to program.

Annotating code to make it possible for automatically converting sequential code to parallel machines is also difficult [13]. Such systems are low-level, and require the programmer do as much work in programming the concurrency as he does in programming the rest of the job. The result is carefully optimized code that is sometimes quite fast, but can be brittle to changes in available processors or faults requiring re-arrangement of data and code.

In our view, we should simply be writing the code that produces the answer, and let the machine do the arranging. Functional code is one way to do that. Such programs run just fine on sequential machines, and when you have available a multi-core machine, it will run even better.

Example A: Program r below is not a function.

```
extern int sum;
int r(int a, int b, int->int f) {
        sum = 0;
        for (int i=0; i<b; i++)
                sum += f(i);
        return sum; }
```

Program r is not a function because the variable sum is computed (assigned) more than once, and the value of r() can depend upon hidden interactions with other programs (even f) via variable sum. This makes it difficult for a person or an analyzer to decide if any given call to r() can run concurrently with any other program in the system. But people do write such programs, and programmers can sometimes annotate or mark a program to help an analyzer find opportunities for parallelism [13]. My position is that it is better just to write the program so that, as much as possible, it behaves in a functional manner in the first place.

Example B: Program m below behaves externally as a function, but internally uses a non-functional element (the memory cell "sum") and must work if f is not a function.

```
int m(int a, int b, int->int f) {
        int sum = 0;
        for (int i=a; i<b; i++)
                sum += f(i);
        return sum; }
```

A graph of program m appears in Figure 2.

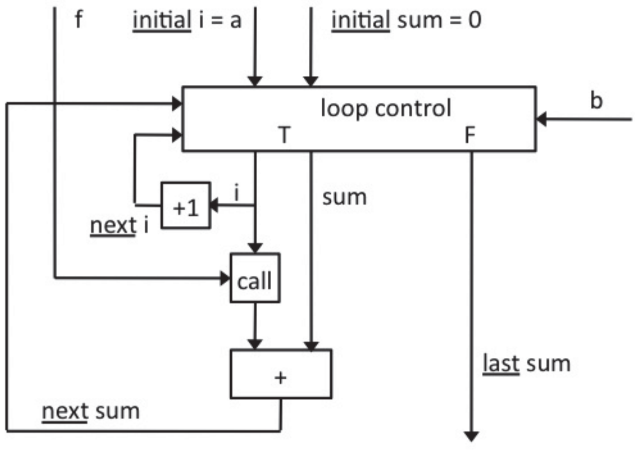


Figure 2 – A graph of program m

If we do m(0,100), program m sums all the f(i) where i ranges from 0 through 99 by re-writing the variable sum. Because f may not be a function, the program sequentializes: the value of each i is computed, then f(i), then the partial sum, and finally we can go on to the next iteration. Since we don't know what f does, the f(i) must execute in this order.

Example C: In a functional system, everything is a function unless declared to be otherwise. Thus f is a function, and we re-write m below as a function.

```
function m(a,b,f) = { s = sequence(a,b);
                      x = map(f,s);
                      return accumulate(x); }
```

Because f is a function, all executions of f are independent. If we run m(0,100), the first statement produces the sequence s = 0,1, …, 99; the second statement uses a function called map that decomposes into two functions mapper and unmapper, where mapper applies f to each value in s, and unmapper gathers the results together to create the new sequence x = f(0), f(1), …, f(99); the third statement sums all the values in x. A graph of the above functional program appears in Figure 3 below.

Note that "map" generates the parallelism, and the sequentiality around the variable "sum" is not present. The functions sequence, map, f, and accumulate are truly functions: no variable is computed more than once (no variable's value is updated), and each function simply produces a result.

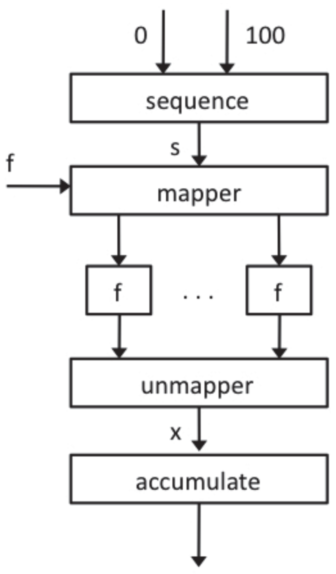


Figure 3 – A graph of the functional program m

An important optimization occurs when we further decompose the above graph into the graph in Figure 4 where, instead of running each call to f(i) on a different processor, the machine groups k function calls on the same processor, with increasing k for smaller f functions. The graph is a little different than one might expect because the sequence function produces the integers using a tree, and the accumulate function sums using a tree. Such an accumulate function assumes that the sequence over + is associative, meaning that the terms can be grouped arbitrarily, though the order of the terms in the sum must remain the same [5]. If your circumstance does not allow you to say + is associative (for example, overflow may occur if sums are not properly grouped), then you cannot use this version of accumulate. Instead, some other version of accumulate would be appropriate.

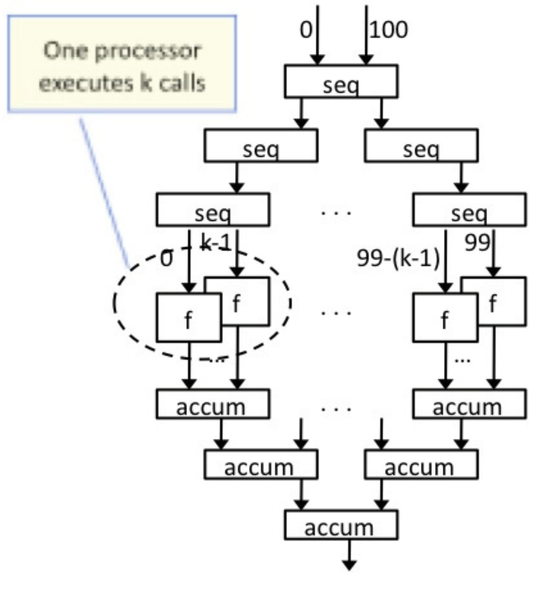


Figure 4 – Further decomposing the graph of Figure 3

A word about granularity: there is no one appropriate level. For some problems, maximum performance is achieved when even simple arithmetic operations such as +, -, *, / are spread over the machine. For other problems, it is at a much higher level. It also depends upon the meaning of "performance": speed, memory usage, power consumption, and so on. For this reason, we define concurrent execution to occur at a function call. Any given function call may be sent to run concurrently elsewhere. The decision as to whether the given call is treated as a point of concurrency or not is made at run-time, even at the time of the call. Reference [15] reviews a number of algorithms and scheduling methods applicable to this machine.

In the above paragraphs, we progressed from a non-functional to a purely functional program, and showed how the resulting code is executes, and adapts, to the current processor situation.

5. The Apply Tree

In this section we will get into the details of how the machine creates concurrent threads at function calls and how it adapts to varying numbers of processors.

Everything begins with a function call f(x)[3] which, inside the machine, is the fundamental box called apply, where apply(f, x) = f(x). Apply's job is to receive the value f (a function) and the data value x, and cause a processor to execute f(x). We augment apply with another input P which is the specific set of processors available to that apply for running f(x), represented in Figure 5 below.

Now f may have several function calls inside it, such as shown below in Figure 6. Note that each of those function calls is an apply and thus has a processor resource input. There are resource allocation functions within f that distribute the processor resources however f desires. A compiler can produce default distribution strategies and associate them with each function f it compiles, but the user can also supply custom methods.

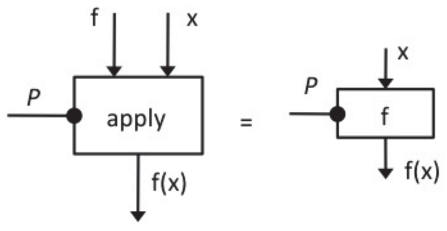


Figure 5 – The definition of function application

[3] To simplify notation, we often show functions as having only of one argument, though we could have written x1, …, xn instead.

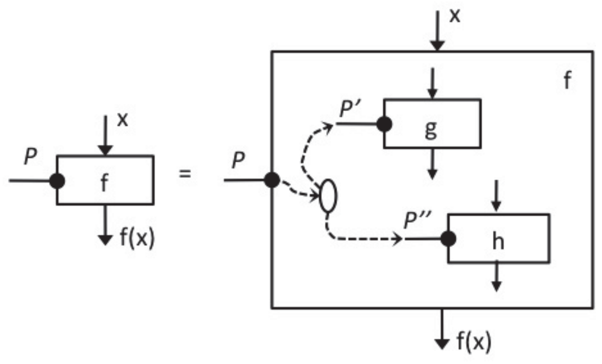


Figure 6 – Function f calls other functions, and sub-allocates the processors.

Finally, should apply decide to run f(x) on another processor, then it sends f and x to a remote apply which we illustrate in Figure 7 below.

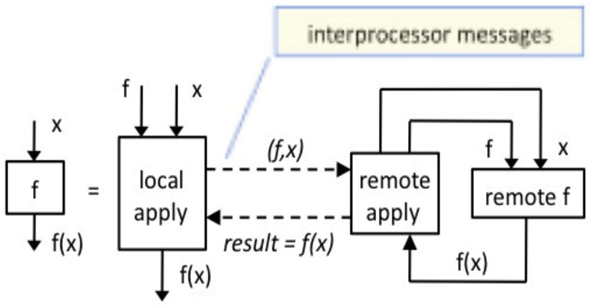


Figure 7 – An apply running f remotely

When apply in processor p spawns a remote apply in processor q, the return address sent to q is (p, local-apply-address). The remote apply in q then sends its result back to local-apply-address in p. Any apply can run on any processor.

The usual execution of nested applies on a single-processor sequential machine uses a stack. But with concurrency we have not just a stack of applies but a tree of applies, all of which can be running at the same time. A simple, bi-directional or doubly-linked apply tree is shown in Figure 8 as it would appear for function f from the example above, were function h to make two additional function calls that run concurrently.

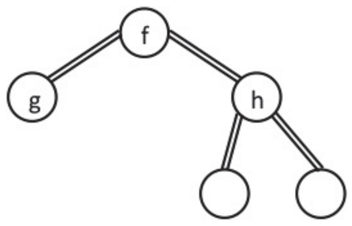


Figure 8 – A portion of an apply tree of function f

You should picture values in the tree flowing down as arguments, and back up the tree as results. The tree is constantly adding and deleting nodes at the fringe as functions call and return. However, each node is actually a thread context rather than a function call frame on a stack, as information (for example, processor assignments) can flow in both directions during the lifetime of any given node. In particular, the P input to each apply (node in the apply tree) is still available and can receive further information regarding processor resources. For example, an overseer function may decide that it wants processor q returned, that is, removed from the pool currently in use. The overseer could begin at any node of the apply tree and tell it to return processor q. In turn, that apply node sends notices to its child nodes, telling them to return processor q. Since nodes are constantly being created and destroyed, eventually all the nodes using processor q will complete, and the parent will simply no longer assign any of its children to use processor q. However, since some tree nodes may persist for extended periods of time, such a passive response to the original request to return processor q may not be satisfactory. In this case, the nodes using processor q need to be moved, and then q can be returned to the overseer.

In this section, we covered in some detail how the machine creates concurrent threads and adapts to varying numbers of processors. Moving computations is discussed in the next section.

6. Moving Functions and Data In Real-time

Consider the Extended Example function m above. Within m, let's start the execution of f(17) and then stop the processor part way through (say the machine detected an error) and move the computation of f(17) elsewhere. Can we simply re-run f(17)? Yes; because f is a function, its only effect is to produce a value – there are no side-effects or actions that modify any variable. Furthermore, restarts require no special work on the part of the programmer. As long as f is truly a function and its inputs are still available, its execution can be replicated, moved, or re-run, and the same result is always returned (or there is an error).

The phrase "moving functions" refers to moving the execution of a function as well as the code itself. Recall that the apply tree is doubly-linked, with both physical and logical pointers. There are actually several methods for moving a node; we describe one of them here: To organize the movement of nodes, we create a "tree mover" agent, whose job is to decide which nodes to move from one processor to another, and to start the moves. The mover starts at one node, moves it, and then moves each of its child nodes, as needed. A node is moved or not moved depending on the purpose of the move, but the mover can propagate as far as needed, and then stop. Because each node is actually thread, the move is done by the thread: the thread notes the fact that it is moving, copies itself to its new destination, re-directs the sources of any messages it expects to receive (and, in the interim, forwards to the destination any inputs that it does receive) waits to hear from those re-directs and from the destination that all is ready, and then destroys itself.

Another type of move causes all nodes from a given program residing on processor P to move to processor Q. Such a mover does not crawl a tree, but rather finds all nodes on P, moves each node to Q, and then quits.

The reader may be of the opinion that there is significant overhead present just in case some function execution is to be moved. There is definitely more work that is done for a given function call, but we submit that the absence of the ability to move functions and data leaves the system brittle and with fewer options for power management and recovery from faults. If we are to achieve truly robust computing, where parts of a distributed machine simply disappear, but the necessary computations continue on, then such overhead is not really overhead – it is essential.

7. Fault Detection, Isolation, and Recovery

In this section we will visit the problem of faults. Please note that our purpose is not to suggest that some particular fault protection and recovery scheme is appropriate for any specific circumstance, only that some schemes are very easy to implement in the proposed system.

Because there are no side-effects, strategies like TMR (Triple Modular Redundancy) and check-pointing can be done with virtually no work on the part of the programmer. For example, if there is a function call f(x) in the program, and the function is to be protected by TMR, then writing TMR(f,x) instead of f(x) executes f(x) in three different processors, and votes the result. Pseudo-code for a simple TMR function is:

```
TMR(f,x) = {
        y1 = apply(f,x);
        y2 = apply(f,x);
        y3 = apply(f,x);
        return  y1 if (y1 == y2 && y2 == y3) else
                y2 if (y1 == y2) else
                y3 if (y2 == y3) else
                y1 if (y1 == y3) else failed; }
```

The programmer (and the system) need know nothing more about f than that it is a function, and the TMR will do its job.

The above shows a simple voter in the TMR. But in a real-time system, more difficult situations can arise. For example, if one of the results is not returned, but the other two agree, then the TMR can reply with the value agreed so far so as not to make the computation late. But the point is that with functions, the job of replicating function execution is simple, and one can see how the system itself could decide to apply TMR to function calls on its own to give a desired level of protection. Any function in any place in the program can be run using TMR without the participation of the programmer. This is not possible in traditional, sequential programs.

Both check-pointing and temporal redundancy are also options, again with little programmer effort. An apply(f,x) can save f and x before spawning. If for some reason it becomes necessary to re-run f(x), the saved f and x are all that is necessary, and can be re-run at any time.

In this section, we discussed how specific fault detection and recovery schemes may be implemented. Again, this is not to imply that more sophisticated schemes might be necessary. But we do claim that the schemes discussed here are much simpler to implement and use in a functional system than in a non-functional one.

8. Non-Functional Programs

It is not difficult to write most programs in a functional style, and as long as significant portions of an application are functions, you will get the benefits when run on the proposed machine. A detailed example appears in [14], but here we briefly address the issue of functional programs and state, such as resources and resource allocators. For example, consider an allocator that receives requests from a user to acquire and release chunks of some resource R. The calls allocate(R, q) and de-allocate(R, q) are requests to R to allocate, and to return, q instances of R, respectively. Please see Figure 9 below.

The allocator simply loops on the current state of resource R and waits for a request (allocate or de-allocate) in the input stream Q. If the request is to acquire q units of R, the allocator decides if it has enough, returns either q or zero units of R to the caller, and correspondingly updates the state of R (stream "num" in Figure 9). The state update is at the end of the loop, after which the allocator waits for the next request. A request to return q units of R is always accepted and results in an increase in the next value in stream "num". The initial value of the loop variable is the number of units of resource that R can allocate.

Curiously, R above is a function if we consider R to be function over a stream of inputs, rather than a function on each item in the stream. That is, given the same input stream Q (the sequence of allocate and de-allocate requests) the allocator R always produces the same output stream – it does not re-write any values. So function R can be written in a functional language, even though the callers of R see it not to be a function. And since any call by a program g to a resource allocator is a non-functional action, the caller g becomes non-functional. Nevertheless, the vast majority of programs remain functional and can be executed as such on the proposed system. But we must make provision for non-functional code (resource allocators or otherwise) and isolate appropriately the non-functional parts.

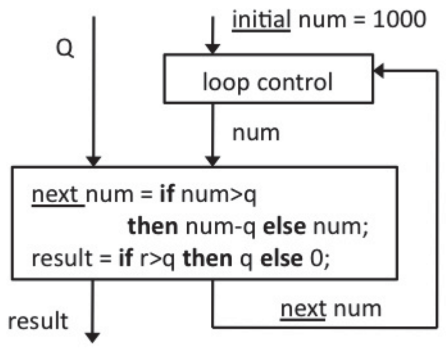


Figure 9 – An allocator as a functional program with a stream Q of user allocation requests

An example of such isolation is using non-functional code with TMR. Assume we have a function f, except that at one point in f there is a single call to resource allocation program h. TMR(f,x) will cause h to be called three times instead of once, and each call to h within TMR(f,x) may produce different results. The solution is that TMR must know that h is not a function, and thus TMR must "unTMR" before the calls to h. unTMR requires that the corresponding calls of h in each of the three f's all be replaced by a single call to h(w). The unTMR(h,w) votes the three instances of argument w, then makes one call to h(w) and returns the same result to each of the f's in TMR(f,x).

Pseudo-code for unTMR is

```
unTMR(g, z1, z2, z3) = {
        z =  z1 if (z1 == z2 == z3) else
             z2 if (z1 == z2) else
             z3 if (z2 == z3) else
             z1 if (z1 == z3) else
             failed;
        return  failed if (z == failed) else g(z); }
```

UnTMR is an example of how non-functional code can be mixed with functional code, and still allow the system to deliver advantages for the functional part. However, note that non-functional code is trouble. It can be accommodated, but it makes the resulting system less flexible, and prevents the system from delivering its full capability. So the goal is to isolate and reduce non-functional code to a minimum.

9. Power-aware Computing

Adjusting power needs to the current compute load has been a part of commercial computing for some time [1], so it is not entirely new to multi-core. But it has come to the fore in space-borne applications both because multi-core can draw considerable power, and because it provides another knob to adjust power consumption to fit the situation. And regardless of how little power one chip may draw, combining hundreds of chips together can make the situation acute [3]. It is always a good idea to use as little power as you need, possibly with some hot spares ready to go to work in case of a problem.

In the system proposed here, each function f (both user functions and library functions) has an associated "load" function load(f) = (a,b) that returns to the caller a measure of the "load units" that function f will present to the system, where a is the load presented by f(x) on a single processor, that is, the total amount of work to be done, and b is the number of processors f could usefully use, or, the degree of parallelism f exhibits. For example, the load function associated with map(f,s) is load(map, (f, s)) = (load(f,1), length(s)). A compiler could produce simple load functions automatically, but handcrafted load functions can replace compiler-generated functions. So when apply(f,x) is about to run f(x) on some processor, apply first runs load(f,x) and uses the result to decide whether to ask the system resource manager for more processors, possibly requesting the processors for f's exclusive use. When asked, the resource manager considers the availability and then returns to apply any additional processors to use. When apply is done, it returns back to the resource manager any processors it received so they can be used elsewhere.

Because it is a functional language, many instances of apply occur in quick succession. So should the resource allocator decide to take back resources, the system will respond to the change in a timely way. When processors are to be removed, apply should engage the node mover to move ongoing computations away from the processors being de-allocated.

10. Summary and Future Work

This paper presents a multi-core hardware and software system emphasizing power-aware, fault-tolerant, and real-time operation. It describes a mechanism for moving on-going computations and data, a capability central to achieving power-aware and fault-tolerant computing. It is based on a functional model of execution (as opposed to the von Neumann model) – a basis that makes simple the behaviors described here. The hardware is simple, as there is no shared memory. This alone frees up considerable silicon resources for other purposes.

Several items are still unfinished, for example, moving a computation is not yet fault tolerant. But after completing these, the next step in the design is to model the system, especially power and fault models, and simulate and evaluate the performance, under normal operating conditions, fault situations, and power allocations.

ACKNOWLEDGEMENTS

Research carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The author also wishes to acknowledge several discussions with Rafi Some and remarks from some of the anonymous reviewers that helped clarify a number of issues.

REFERENCES

[1] Ranganathan Parthasarathy "Recipe for Efficiency: Principles of Power-Aware Computing" Communications of the ACM vol. 53 no. 4 (April 2010).

[2] Intel, "Single-chip Cloud Computer" Intel Labs Single-chip Cloud Computer Symposium, March 16, 2010 http:www.intel.com/go/terascale

[3] Peter Kogge, "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems" Univ. of Notre Dame CSE Department TR-2008-13, September 28, 2008.

[4] Peter M. Kogge, "EXECUBE-A New Architecture for Scaleable MPPs," International Conference on Parallel Processing (ICPP '94), 1994, vol. 1, pg 77-84.

[5] Guy L. Steele, Jr. "Organizing Functional Code for Parallel Execution; or, foldl and foldr Considered Slightly Harmful" Invited Talk at the International Conference on Functional Programming (ICFP) ACM SIGPLAN Edinburgh 2009.

[6] The Fortress Language Specification, v1.0. Tech. Rep., Sun Microsystems Inc., March 2008; or http://projectfortress.sun.com/Projects/Community/wiki/FortressDocumentation

[7] Michael Malone, "Onboard Processing Expandable Reconfigurable Architecture (OPERA) Program Overview" Fault-Tolerant Spaceborne Computing Employing New Technologies Workshop, 29 May 2008.

[8] John A. Sharp ed. Dataflow Computing Theory and Practice Ablex Publishing 1992.

[9] Arvind and Kim P Gostelow "The Id Report: An Asynchronous Language and Computing Machine" Tech Report TR-114 Dept of Information and Computer Science, Univ. of California, Irvine Sept. 1978.

[10] Jack B. Dennis, "General parallel computation can be performed with a cycle-free heap" Proceedings of International Conference Parallel Architectures and Compilation Techniques, 1998, pg 96-103.

[11] Bryan O'Sullivan, Don Stewart, and John Goerzen Real World Haskell, O'Reilly, November 2008.

[12] "MPI: A Message-Passing Interface Standard" Version 2.2 Message Passing Interface Forum September 4, 2009.

[13] Joseph A. Roback and Gregory R. Andrews "Gossamer: A Lightweight Programming Framework for Multicore Machines" Proc. Second USENIX Workshop on Hot Topics in Parallelism (HotPar '10) Berkeley, CA June 2010.

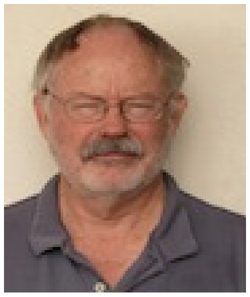
[14] Arvind, Kim P. Gostelow, and Will Plouffe, "Indeterminacy, monitors, and dataflow". ACM SIGOPS Operating Systems Review vol. 11, no. 5 (Nov. 1977).

[15] J. Liu, and VA Saletore "Self-scheduling on distributed-memory machines" Supercomputing '93 Proceedings 1993 pg 814-823.

[16] William Blume, *et al*, "Polaris: Improving the Effectiveness of Parallelizing Compilers" in Languages and Compilers for Parallel Computing, Springer Lecture Notes in Computer Science 1995, vol. 892/1995, pg 141-154.

[17] Peter Kogge "The Technical Challenges of Extreme Scale Computing" Technical Presentation, Jet Propulsion Laboratory May 7, 2010

BIOGRAPHY



Dr. Kim P. Gostelow *is a flight software engineer at JPL, currently working on the Mars Science Laboratory missions. He has also done flight software for the Mars Pathfinder and Mars Exploration Rover projects, as well as the Cassini mission to Saturn, and others. Previous to working at JPL, Dr. Gostelow was on the faculty at the University of California, Irvine, Information and Computer Science department, where he did research on dataflow computer systems and programming languages.*