Position: Flexibility of Runtime Support Beats Specific Parallelism Construct Support

Sean Halle
Open Source Research Institute
Email: sean.halle@osri.org

***Abstract*—This is a position paper, to provide food for thought and debate. Even so, the ideas are extrapolated from published work on runtime systems and hardware abstractions that have been implemented and successfully demonstrated.**

To bring parallel programming into the mainstream, it needs to be productive, source code must port easily with high performance, and parallel programming has to be favorable to industry for adoption. In previous work, we took the position that to attain all three, software should be organized into a stack, based around *specialization* of source to target hardware. Each layer of the stack has a role in the specialization process, which spans an application's lifetime from transform to hardware-specific form, to installation, to runtime. In this view, specialization includes the toolchain, hand-tuning, auto-tuners, multi-kernels, profiling, and binary optimization. Here, we briefly restate the elements of such a stack, and how it encapsulates and organizes these.

If the premise of such a stack is accepted, then in this paper we take the position that hardware should support tightly integrated *firmware*-based runtime systems rather than specific parallelism constructs. This is a new category of firmware that is tightly integrated into the processor pipeline and managed by the OS. We describe hardware structures that support such firmware, and allow traditional thread constructs, domain-specific constructs, transactional memory, and even consistency models to be implemented via such firmware. Such constructs have extremely low overhead, as well as engage the language runtime into pipeline-level hardware resource management.

I. Introduction

Current parallel programming is blocked from mainstream industry because it has lower productivity than sequential programming, forces a rewrite of source for each new target to get good performance, and disrupts the ways programmers think and their workflow. All of which makes it too expensive.

Many believe a solution to productivity is domain-specific languages. However, to be a real solution, a large number of such domain-specific languages have to be created and ported to each hardware target. Such creation and porting must be done inexpensively due to the small user base of a language.

Solving performant-portability is more difficult. Such portability means source is written once, then automatically specialized to all hardware targets, so that it runs high performance on each. To achieve this, the one source has to capture all information needed by all specialization techniques for all hardware, current and future.

Adoption by industry is the least research-oriented aspect, but for parallel programming may be the most important. To be adopted, a solution would have to be flexible enough to support all the domain-specific languages, fit with any of the array of programming styles and workflows used in the industry adopting, and offer smooth transition from current programming practices to the new ones. It has to seamlessly work on both current hardware and future parallelism-supporting hardware.

We term this the triple goal of: productivity, performant-portability and adoptability for parallel software. Throughout the paper, we tie specific details of our proposed approach to these three goals.

One suggested solution to the triple goal is a software stack that is based around specialization, and collects independent, small, contributions to the stack, which collectively perform the specialization process[3]. Productivity is solved by efficient and practical support of domain-specific languages. Performant-portability is solved by conveniently supporting the full range of specialization techniques, and accumulating them from many sources. Adoptability is solved by flexibility to adapt to current and future hardware, with gentle transition that is practical, cost-sensitive, and effort-reducing.

In this paper, if the premise of such a software stack is accepted, and the premise that domain-specific languages solves the productivity problem, then we propose that supporting runtimes in hardware is better than supporting any particular set of parallelism constructs, even ones as basic as the Compare And Swap instruction or Thread constructs.

The reason is that specific constructs have nonuniform performance when taken across languages, and so fail to support domain-specific languages. Such hardware will perform slightly better than the proposed firmware approach on the programming models that fit the hardware, but worse on all others. This is in conflict with domain-specific languages, which by their nature encompass a wide variety of constructs, most of which won't execute well on the direct hardware.

Specific constructs in hardware also have a chicken and egg problem because they only give advantage for a few specific languages. Unless those languages are dominant, the expense of commercializing such hardware can't be recovered. And without a hardware support advantage, it is unlikely for a subset of *parallel* languages to gain such wide dominance.

In Section II we give details of the hardware we propose to support firmware runtimes. In Section III we expand on the software stack and how it fits with the runtime hardware and how the two together support the three goals. In Section IV we apply the proposal to the topics of interest of this workshop to see if they are consistent and address the concerns. We conclude in Section V.

II. What parallel abstractions should the hardware provide?

Our position is that the hardware should not directly supply parallel abstractions. Instead, it should supply a mechanism that elevates the language runtime to a soft-extension to the instruction set, making the runtime separate from the executable and separate from the OS. Thus, parallel abstractions are implemented as firmware that extends the hardware. With suitable support, many firmware-implemented parallel abstractions would require only a handful of instructions with a similarly low number of cycles of overhead.

This arrangement solves a number of problems currently facing language designers and runtime implementers. First, it makes all application-resident information available to the runtime, and gives it control over the innermost level of hardware, right down to swapping contexts in and out of registers. Second, it increases practicality of domain-specific languages, which is one main path to high programmer productivity. Third it improves portability directly, and fits the proposed software stack arrangement, providing a natural and smooth transition from existing hardware to hardware with such firmware runtime support.

A. Soft-extension of instruction set

Precedence for soft-extensions to instruction sets exists. The Alpha chips from DEC executed complex VAX instructions by switching fetch over to a special memory containing normal Alpha instructions, which implemented the functionality.

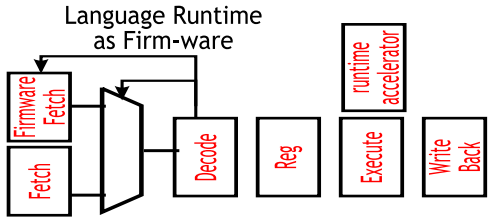

Fig. 1. A special `switch` op-code is recognized by the decode stage, and triggers fetch of instructions from firmware. The firmware instrs are linked by the OS and implement the runtime behavior of a language. Helper instructions accelerate common runtime operations, such as hash table lookups, communication calculations, search for optimum, and so on.

An analogous approach is illustrated in Figure 1. Here, one op-code is set aside as the "switch to runtime" operation. Its execution causes instructions to switch to fetching from the firmware. Information is communicated via register contents, which point to data-structures that include a hardware defined portion and a language defined portion.

This firmware was written by the language provider, so it is separate from the executable. It implements the behavior of parallelism constructs of the language.

Such an approach addresses security, portability, and efficiency. It is secure because the OS controls the firmware. It is portable because the executable only contains the *interface* to the constructs (implementation is separate). It is efficient because the firmware runs in user-space, and switching to it costs the same as a `call`. This also improves application performance, because a firmware runtime has control over low level behaviors such as hardware-supported swapping of contexts and control of hybrid cache/scratchpad memory.

The firmware runtime receives application information in the data-structures, such as construct semantics and information extracted by the toolchain for the runtime. The firmware runtime uses the application information to control swapping execution contexts, initiating communications, and any other resource management.

Portability improves because only the *interface* to constructs is encoded in the executable. Implementation is free to change from one processor to another, or even from one level of a machine's hierarchy to another.

B. Communications via firmware

By removing communication from the executable and putting it into firmware, both a portability benefit and a control over hardware benefit are gained.

The portability benefit is realized when firmware becomes the application gateway to communication. This lets parallelism constructs be application oriented, merely implying communications, without specifying or controlling communication details, which invariably imply hardware details.

The control benefit results when firmware controls activities such as marshalling data and invoking the hardware to communicate it. The firmware runs in user-mode but is trusted with hardware resource control, so it can do things like make communication events trigger suspend and resume of tasks.

C. Communication via separate helper processors

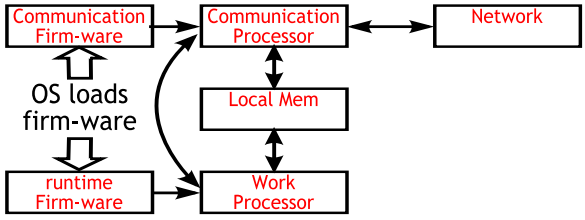

Fig. 2. Communication is performed between local memory and remote memories via a separate communication processor. This processor executes firmware that is loaded under OS control. For example, it may run a standard software cache or run scatter-gather code extracted from the application.

Controlling communication inside the firmware improves the practicality of adding separate helper processors for communication. These would overlap communication overhead with useful computation, as illustrated in Fig 2. These processors execute separate firmware, supplied either by the OS, or as part of the executable.

A cogent example is an application with complex data structures that are communicated between long running tasks. During a task, some portion of the data-structure is bundled up and sent to another task.

The language used provides constructs for rendez-vous style send and receive, plus constructs that identify the bundle-data and unbundle-data code. Send and receive are implemented

as part of the language, as runtime firmware. In contrast, the bundle and unbundle code is extracted from the application by the toolchain and packaged into the executable. During the run, an OS call causes that bundle and unbundle *communication* firmware to be linked into the communication processors.

When a task executes send or receive, the runtime firmware swaps the context out, suspending the task, and replaces it with a non-blocked task. Simultaneously, the runtime causes the communication processor to execute bundle or unbundle code. When communication completes, the task is unblocked.

This tight integration of communication with scheduling of tasks is an example of application information driving scheduling. It allows the firmware to decide which core to assign a task to based on application code and input data, while maintaining ultra low overhead.

Such bundle/unbundle doesn't work as well in cases where the data consumed has little predictability, or the application doesn't provide gather-scatter or bundle-unbundle information. In this case, the OS can link standard software-cache firmware into the communication processors.

Such a cache has the advantage of being able to swap out tasks when it misses, which gives an efficient way to overlap cache misses with useful work, without the area and energy overhead of out-of-order pipelines. This requires the hardware to make the cost of switching tasks be like that of a function call, and the application to supply sufficient parallelism.

Another potential advantage is adjusting the cache characteristics during the run to better match the application. The characteristics can be measured, or the toolchain can insert the results of analysis.

This would ideally be coupled with scratchpad memory that can treat a section of memory as tags. The communication processor is given control, to configure the tag memory, to cause tag comparisons, etc. Previous work suggests that such a software cache compares favorably with hard-wired caches[2].

D. Speculation and Fast Control Message Support

Hardware support for speculation will work especially well with a firmware runtime coupled to a communication processor. Transactional memory, thread level speculation, and higher level speculative constructs could each be supported by generic lower-level mechanisms, which are in turn invoked by the communication firmware, as hinted at by Carter[1].

This arrangement isolates hardware from the language consistency model and execution model. There should no longer be a large penalty for mismatch. To get this decoupling, hardware is simplified, by factoring the semantics out.

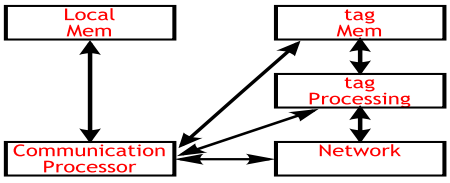

Fig. 3. Tag memory and tag processing are added to local memory. The tags have an extra field used by tag processing to filter lines.

Fig 3 shows a step towards such a refactoring. Hardware support is added in the form of tags plus processing, which can be used for check-pointing, sand-boxing, and speculative tie-points. They don't imply application visible semantics. Rather, they are used inside the firmware to implement semantics like transactional memory, thread-level speculation, and consistency models such as acquire-release, or flush-on-command.

For example, for check-pointing, the tags are just as in caches, but an additional field holds a check-point number. Writes are only performed to lines with the same check-point number, and if none exist, a read is performed, of either the most recent previous check-point or fresh from remote memory. The hardware supports sending and comparing lists of lines with the same check-point number, as well as sending only lines from a particular checkpoint. This efficiently supports Thread-Level Speculation, with rollback and commit.

Sandboxes use the same hardware, except instead of storing the check-point number, the extra tag field holds the sandbox ID. For transactional memory, each transaction started gets its own sandbox ID. This supports the TCC style transactional memory implementation[6].

Checkpoints may also be used to support shared-memory style consistency models, but speculatively. New check-points are periodically generated, while previous ones are examined for conflicts. Examination takes place in the communication processors, supported by hardware for comparing lists of tags. Conflicts cause rollback and restart, with updated state from one of the conflicting local memories.

Such hardware can also be used to turn off the tight consistency of current snooping based protocols for the bulk of computation, saving time and energy for the code that doesn't need it. Such consistency is only enabled for a few specialized portions of code, those that use shared variables as control messages, such as in software based mutex algorithms.

Another alternative is to only update shared memory when synchronization constructs imply handoff of ownership. This uses the tag hardware to track individual objects or data structures. The synchronization construct in the runtime firmware triggers the communication firmware to update all objects on the core that is gaining ownership, from modifications made on the core giving up ownership. The tag-processing comparison functions make this fast and efficient. This not only eliminates the time and energy lost to snooping and directory protocols, but also simplifies the programming model and removes nonportable shared-memory code from executables.

These approaches rely upon having fast control messages that communicate things like lists of tags. Fast control messages allow each core to have its own runtime, with purely local state, which is the highest performance runtime approach. They use the high speed control messages to communicate constraint updates, explicitly send tasks to load balance, etc.

Such internal-to-runtime messages have only small amounts of data, but their latency is crucial to the runtime's responsiveness. A slowly responding runtime will leave its core idle more often, because the rate of handling internal bookkeeping about tasks is slower than the rate of finishing those tasks. It is in

this case that fast control messages become crucial[8].

E. Example

To illustrate such hardware in action, we walk through an application binary as it invokes the "acquire mutex" parallelism construct:

a) setup and switch: At the appropriate place in the binary, instructions load one register with the pointer to a mutex structure, and another register with the pointer to the virtual processor (VP) requesting the mutex lock. Next, the `switch` instruction executes, which switches fetch over to the firmware of the runtime, while saving the stack and frame pointers into the data-struct of the requesting VP.

In this example, the hardware specifies a "virtual processor" (VP) data structure. The first locations make up a hardware defined portion that the `switch` instr automatically manages.

b) runtime internals: After `switch`, runtime code executes from the protected firmware. The code for mutex acquire expects a pointer to a mutex struct to be in a particular register, checks the "current owner" field, and if empty writes the pointer to the VP (held in another register) into it. It then marks the VP as unblocked. Similarly, if the mutex is already owned, it places the VP into the mutex struct's queue, where it remains blocked.

Most importantly, if the mutex is already owned, the runtime swaps the requesting VP out from the hardware context (register set). It swaps in an unblocked VP.

The execution time of this can be on the order of 10 cycles. Such speed requires hardware support for swapping VPs in and out, such as set aside cache or scratch-pad memory with a wide port to registers, and speculative access to the mutex data-structure. This makes all memory access local and fast.

Speculative accesses would be verified while computation continues. If memory consistency is performed only upon command of the runtime, and communication hardware supports check-point and rollback, then application work can continue without speed penalty, except in case of rollbacks.

Notice that no atomic memory instructions have been used. Further, the application binary contains only *interfaces* to high level constructs. All operations have been local and fast, despite maintaining global consistency of global address space.

III. Which should be the responsibility / functionality of the programmer, the runtime software, and the hardware?

According to the cited work on portability, responsibilities naturally break down along the lines of a software stack[3]. The goal of which is to support specialization, the process of transforming the original source into a form that is highly efficient on the target hardware. This is the heart of portability.

Each layer of the stack has some role in the specialization process, while the application, on top, provides the information that the rest of the stack needs while performing the specialization. Ideally, the application must not expose hardware assumptions nor hinder specializations for particular targets.

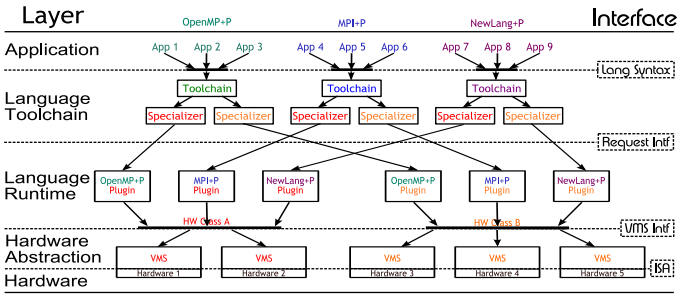

Fig. 4. The layers of the previously proposed software stack.

The layers and interfaces are seen in Fig 4. On top are the applications, which use constructs to capture specialization information used in the rest of the stack. Languages make the interface between applications and the toolchains that generate the executables. The most important interface is between the executables and the runtimes. With this proposal it will be the `switch` instruction plus hardware defined and language defined data-structures that get passed between. The runtimes rest on top of a layer of hardware abstraction implementations, which exports an interface that simplifies runtime creation. The abstractions are then implemented in terms of the Instruction Sets of multi-core chips.

Productivity is solved by making domain-specific languages simple to create, easy to port across hardware, and high performance. The application programmer only sees application relevant concepts, reducing their learning curve and matching their mental model to the language. The domain-specific parallelism constructs can be provided either embedded-style as library calls, or with compiler support.

For portability, the languages also design their constructs to avoid hardware implications. Languages that have succeeded include CnC[7], WorkTable, and HWSim. Such constructs are implemented mainly by the runtimes, and occasionally by the toolchain. Using such constructs doesn't by itself ensure portability, but it goes a long way towards that goal, by removing the largest source of hardware-specific information.

Applications on top of such a stack should not use shared variables without protecting access via a language construct. This precludes "roll your own" synchronizations, or communications implemented in the app by using shared variables.

The proposed hardware naturally supports this stack. The hardware abstraction, used to simplify runtime creation, is currently implemented as a software layer, including assembly primitives for switching between application and runtime. The proposed hardware itself implements much of this abstraction.

Large portions of the language runtime code that currently exists for multi-cores should work verbatim with the new hardware support. Only portions that take advantage of acceleration should need modification.

This helps adoptability of the new hardware, by providing a seamless migration from current hardware to the new, without modification of application code. And, being able to repurpose

runtime code to the new hardware also eases adoption of it.

Contrast this with hardware that directly implements specific parallelism constructs. The abstraction can still be supplied for it, but the construct hardware is only used when running code written in a few languages. It fails to equally support domain-specific languages, and so harms productivity.

At a minimum, specialization needs constructs that identify the tasks, the constraints on scheduling the tasks, and the data to be communicated between tasks. However, high quality specialization requires additional "helpers" [3]. These enable: 1) modifying the layout and order of access of data, 2) modifying the size of a task, both the data consumed and code executed by it, and 3) predicting both execution-time and data consumed by each task. An example is DKU[4], which provides task-size-modification helpers.

Helpers related to data consumed by a task and layout directly feed into communication firmware. Either the toolchain generates firmware from construct semantics, or constructs identify the application code to be used as communication firmware. Domain-specific constructs must be designed to capture the information, and convenient for the tools to extract.

One last concern is the creation of the many firmware runtimes. The hardware abstraction interface must uniformize them, to reduce the work of creating one for a particular language. An example abstraction is Virtualized Master-Slave[5].

IV. Specific Topics of Interest

Now that a position has been stated, let us see how it applies to the topics of interest, to check consistency and usefulness.

c) enabling future parallel programming models: Essentially all current and foreseeable future parallel programming models should be supported in a fairly uniform way. The stack approach makes creating new models fast and easy. Embedding the switch mechanism in the pipeline, and supporting common runtime constraint management and assignment operations like hash tables and context swapping ensures low overhead. In addition, bringing application information into the lowest hardware level of resource management enables high performance scheduling.

d) innovative architectural execution models: Innovative architectural execution models are more practical when isolated from the programming model. The `switch` instruction provides this decoupling, and gives hardware freedom to explore, without code legacy constraining it. However, high speed internal-to-runtime messages, speculation support, and decoupled communication processors may be considered elements of an architectural execution model we advocate.

e) novel memory hierarchies: One suggestion is coupling memories to their own communication processor that performs all movement of data to remote memories. Another is make memories be configurable, with tag fields and hardware to support sending lists of tags that have a given ID, and checking tags against such a list. These support transactional memory, thread-level speculation, acquire-release, and speculative variants of sequential consistency, with the overhead overlapped.

f) simplified and scalable memory models: A wide variety of memory models can be implemented in firmware with the proposed speculation hardware, including simplified high level models implied by domain-specific constructs. Speculation and the linkage to context-swapping allows memory consistency overhead to be overlapped with work. Scalability is then inside the communication firmware algorithm.

g) high level constructs for on-chip communications: Essentially any high level communication construct can be implemented in firmware of the communication processors. Further, linkage between communication processor and work processor brings pipeline-level hardware control to the high level communication constructs.

h) future directions in programming massively parallel systems: We believe future algorithms should divide data and computation in tasks into a fractal-like hierarchy. Each level of task should look the same in terms of communication and computation activity, so that task communication scales as availability in the hardware does, as the hierarch is traversed.

This means programmers need to find hierarchical approximations to problems, where a task in one level accumulates results of lower levels. The stack enables a hierarchy of runtimes that matches hardware and application hierarchies.

i) potential bottlenecks for future parallel systems: We believe the amount of parallelism in code will be the bottleneck. Communication-to-computation ratio of hardware is worsening, and memory size is growing more slowly than computation rate or hardware parallelism. Hence weak scaling doesn't apply. The code has to change, to find smaller work-units within it, else amount of parallelism will be the bottleneck, leaving processors idle.

The stack makes such code performantly-portable.

V. Conclusion

The paper has supported the position that hardware should support firmware runtimes instead of specific parallelism constructs, by showing the benefits of low level hardware management being brought into user-space with an OS managed but language supplied runtime firmware.

References

[1] J. B. Carter, C.-C. Kuo, and R. Kuramkote. A comparison of software and hardware synchronization mechanisms for distributed shared memory multiprocessors, 1996.

[2] Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *DAC*, pages 416–419, 2000.

[3] Sean Halle. PStack home page, 2012. http://pstack.sourceforge.net.

[4] Sean Halle and Albert Cohen. Leveraging semantics attached to function calls to isolate applications from hardware. In *HOTPAR '10: USENIX Workshop on Hot Topics in Parallelism*, June 2010.

[5] Sean Halle and Albert Cohen. A mutable hardware abstraction to replace threads. *24th International Workshop on Languages and Compilers for Parallel Languages (LCPC11)*, 2011.

[6] Lance Hammond and et al. Transactional memory coherence and consistency. ISCA '04, pages 102–.

[7] Kathleen Knobe. Ease of use with concurrent collections (CnC). In *HOTPAR '09: USENIX Workshop on Hot Topics in Parallelism*, 2009.

[8] Chao Mei, Gengbin Zheng, Filippo Gioachin, and Laxmikant V. Kalé. Optimizing a parallel runtime system for multicore clusters: a case study. In *The 2010 TeraGrid Conference*, pages 12:1–12:8, 2010.