The von Neumann architecture is due for retirement

Aleksander Budzynowski   Gernot Heiser
NICTA and University of New South Wales
Sydney, Australia

Abstract

The processor industry has reached the point where sequential improvements have plateaued and we are being flooded with parallel hardware we don't know how to utilise. An efficient, general-purpose and easy-to-use parallel model is urgently needed to replace the von Neumann model. We introduce and discuss the *self-modifying dataflow graph*, an unusual model of computation which combines the naturally parallel dataflow model with local graph transformations to eliminate the need for a global memory. We justify why it is a promising candidate.

1 Introduction

The von Neumann bottleneck would have brought processor performance to its knees many years ago if it weren't for the extensive cache hierarchies used on modern processors to reduce accesses to main memory. Backus [3] argued that it was holding back programming languages. We think it is now holding back adoption of parallel hardware. Even if cache coherence protocols can be made to scale further, constantly moving data long distances on the chip will consume considerable energy and time. The difficulties of writing efficient and correct software with nontrivial sharing [9] are further evidence that the von Neumann model is inappropriate for many-core hardware.

Since hardware improvements are increasingly going to take the form of parallel resources rather than faster sequential computation, we should be looking for a suitable replacement model—not a framework for parallel programming like MPI, OpenCL or MapReduce, but a path that lets code written in any programming style benefit from increasingly parallel hardware.

Due to major changes in hardware technology, the von Neumann design no longer plays the role of computer architecture but rather is a simplified model of the machine, allowing programmers to write software without an intimate knowledge of the particular hardware it will run on, and allowing hardware designers to make their implementation as efficient as possible without needing to consider every program that might be written for the machine. Valiant [18] calls such models *bridging models*, and argues that a suitable bridging model is necessary to help us make a real transition to parallel hardware. Other authors have used "thin middle" [11] and "simple abstraction" [19] to refer to similar things.

Any bridging model must be efficiently implementable, general-purpose and reasonably easy to use. We believe that a good bridging model must have some notion of space, because as parallel machines are scaled up, the difference between sending a message to a neighbouring core and a distant one becomes significant. We also think that a bridging model should be able to run software written in any programming style.

In this paper we present the *self-modifying dataflow graph*, which we believe meets these requirements. One may feel a little threatened by the fact that the model has no program counter, nor `load` or `store` instructions, but perhaps this is the leap we must take to free ourselves of the von Neumann bottleneck. In later sections we explain how the model supports dynamic data structures without requiring a global memory, and also discuss considerations relevant to OS implementation.

2 The Self-Modifying Dataflow Graph

In the dataflow model of computation [8], a program is represented as a graph of instructions, as shown in Figure 1, and the machine executes the graph directly. Data words (*tokens*) flow along the edges like packets along wires. Most instructions are typical mathematical operations like `add`, `subtract` and `compare`; these instructions *fire* when they have a complete set of inputs: this consumes the input words, applies the specified operation, and produces output words (shown in Figure 2).

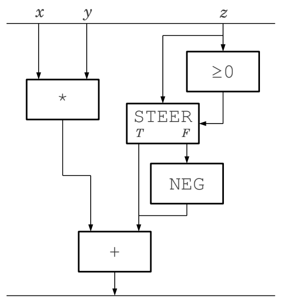

Figure 1: A flow graph for `x*y + abs(z)`.

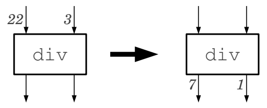

Figure 2: A flow graph instruction before and after firing.

The `steer` instruction transfers the input word to a chosen output edge, allowing branching behaviour (shown in Figure 1). Since an arbitrary number of instructions may fire at the same time, the model is inherently parallel. However, ready instructions may take an arbitrary amount of time to fire, so any execution, from completely sequential to maximally parallel, is valid. To a large extent this means a dataflow program can be written without regard for the number of processors it will be executed on (and the number of processors could even change at runtime).

Dataflow is a naturally parallel model, with different sorts of parallelism exhibited by particular graph structures. Dataflow makes no real distinction between instruction-level parallelism (ILP) and thread or task parallelism. Data parallelism exists where several copies of a code graph can operate in parallel. Pipeline parallelism exists where successive elements of a list or stream are processed by different stages of the computation at the same time.

Unfortunately dynamic data structures cannot be efficiently represented in the basic dataflow model, and neither can dynamic behaviour, such as function calls, function pointers, code-as-data, and pass-by-reference. Previous attempts to overcome these limitations typically involved assuming a shared global memory and either adding `load` and `store` instructions ([5, 15]) or introducing other primitives which took advantage of the global memory (such as I-structures [2]).

Any model which depends on a global memory is going to suffer from a von Neumann bottleneck. We propose an alternative: By introducing a class of instructions into the model whose effect is to perform local graph transformations (both on data and on the program itself), dynamic behaviour can be expressed. We call this model the *self-modifying dataflow graph* (SMDG). The new instructions would be able to create new nodes, and rewire the edges in their immediate neighbourhood. In this model, a data structure, for example a binary tree, need not be a dead representation laid out in a computer's memory: it can be alive and spread across a fabric of processing nodes, just as the *active data structures* concept in the Connection Machine [6]. This is likely to open up opportunities for parallelism on dynamic data structures.

Pure dataflow does not require a notion of global state, and we believe we can maintain this powerful property when enhancing it with graph transformation instructions. Furthermore, we think the model might provide another extremely useful property: The edges between nodes in a dataflow graph are essentially pointers. If the instruction semantics guarantees that arbitrary pointers cannot be constructed (i.e. no casts from int to pointer), then pointers can be used as *capabilities* without any security overhead. This allows a large class of bugs to be ruled out (such as buffer overflows) as well as creating the possibility of a communication channel between programs without danger of the programs modifying each other. Such a communication channel can consist of many parallel data flows.

One more feature completes the picture: a decentralised accounting system for node allocation, for example, instruction support for unforgeable *allocation permits* which can be converted into new nodes and reclaimed on deallocation. This removes the need for running the OS in a special processor mode: all programs in the machine can execute the same instruction set, but may be given different rights through capabilities and allocation permits. This allows an arbitrarily deep nesting of sandbox environments, be they hypervisors or operating systems or sandboxes for untrusted plugins, with barely any overhead: security is guaranteed not through hardware checks but through the set of possible instructions.

Incorporating these properties while retaining a useful instruction set is challenging, and there are going to be interesting and difficult trade-offs. We have designed a simulator to allow us to experiment with prototype instruction sets, and so far implemented data structures including a tree, stack, queue, and deque. We have also built a simple compiler which, itself a SMDG program, can stitch together a simple SMDG program. We are currently working on compiling STG code (an intermediate program representation used in the Glasgow Haskell Compiler) to an SMDG instruction set. Our positive results so far at least demonstrate that the SMDG supports dynamic data structures and other dynamic behaviour.

We give an example from one of our prototype designs to illustrate the graph transformation process. Assume

```
struct instruction {
    int opcode;
    word_t inputTokens[2];
    word_t outputTokens[2];
    struct instruction *output[2];
 };
```

Figure 3: Logical SMDG instruction format

the instruction format shown in Figure 3. The input and output tokens can be data or pointers. The instruction pointer outputs define data flow, i.e. `output[0]` specifies which graph node `outputToken[0]` flows to.

Figure 4 illustrates a list insertion. Boxes indicate graph (and machine) nodes, black arrows indicate data flow (i.e. `output` references), tokens are indicated by blue dots travelling along dataflow arrows, and a blue arrow originating at a token indicate that the token is a pointer and shows its destination. Labels in boxes indicate the instruction the graph executes at a particular node. Labels in italics are comments indicating the purpose of a node which is used just for holding data (i.e. is not executing, no tokens are sent to it).

There are two graph manipulation instructions: `duplicate` creates a new pointer and `writeEdge` creates a new data flow link between the nodes indicated by the instruction's inputs.

In the figure, the top bottom nodes ("template" and the nodes pointed to it) represent the list data structure, while the top two nodes represent the code of the insert operation. To insert an element at the head of the list, a pointer token, pointing to the list head, is sent to the code (top). The `duplicate` instruction duplicates the node pointed to by its input, by copying the instruction (including its references) to a different node, and outputs the new pointer (middle). The `writeEdge` completes the insertion by creating a new link between the nodes indicated by its inputs (overwriting the existing link from the *template* node, bottom).

3 Implementation considerations

Bridging models leave many implementation details open. Initially we plan to implement a software runtime layer that spreads SMDG programs over a multicore or multiprocessor system, but future hardware architectures could be designed to natively execute SMDG programs. Below we describe a reference implementation which justifies that the SMDG can be implemented, and gives us a reference point for discussion.

We view the hardware as an array of processing cores, each with some local memory and communication links

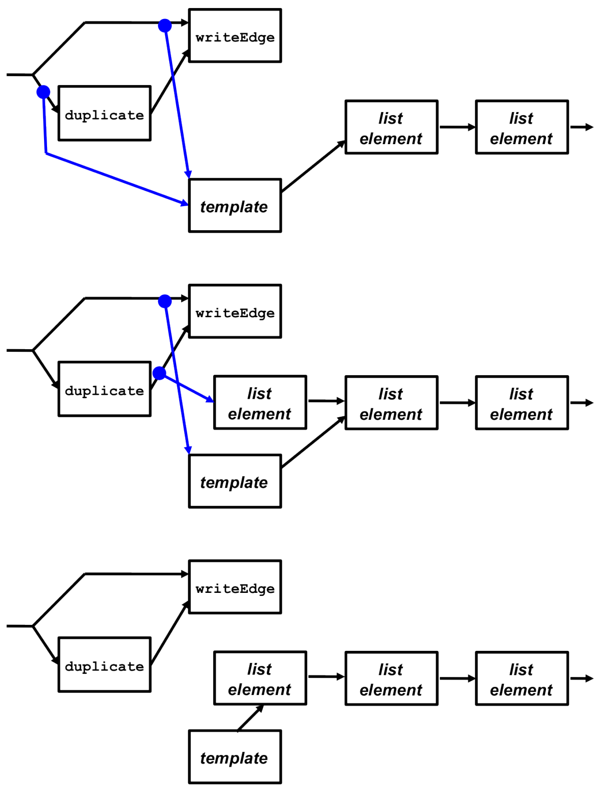

Figure 4: List insertion in the SMDG.

to its immediate neighbours. Such hardware is easy to scale up: adding more core adds more computational power, memory, and communication channels in equal measure. This architecture is a reasonable approximation for most modern parallel hardware, where typically each core possesses local memory (or cache) and communication with nearby cores is faster than with distant cores.

Each core stores a list of constant-sized graph nodes (i.e. instructions) in its local memory, and also space to store tokens (data words) which are waiting (since we use a single-token-per-arc model, the possible number of waiting tokens is bounded). Each instruction contains pointers to its dependents, as well as back-pointers for each incoming pointer, which assist with management. Each core contains a free list to allow new nodes to be allocated. The core may perform analysis on its local graph nodes to schedule them efficiently on its functional units (similarly to traditional ILP) and perform other internal optimisations such as coalescing groups of nodes to reduce bookkeeping. These optimisations are important for SMDG implementations to achieve reasonable performance on sequential code.

If pointers are opaque to software, as discussed in the previous section, then relocating instructions at runtime (to balance work more easily between cores or to reduce distances between dependent instructions) can be performed transparently and safely. We face the difficult problem of arranging computations to minimise communication delay between dependent computations (tending to keep them close together) and execute as many independent computations in parallel as possible (tending to spread them apart). An optimal assignment is NP-hard [4] so we must instead look for good heuristics. Since the graph transformation operations change the graph structure at runtime, an offline algorithm is unsatisfactory.

In our reference implementation, we propose a combination of frequent local and occasional global optimisations. The local optimisations might model the graph edges like springs (tending to pull dependent instructions close to each other) and core utilisation like pressure (tending to balance the load between cores). To avoid getting stuck in local minima, simulated annealing could be occasionally performed, particularly after major changes to the graph (such as loading a new program). On-line simulated annealing was suggested and some advantages discussed in [14]. Recording profile information, such as how often an instruction fires, will allow the allocation to be further improved.

The combination of opaque references with mobility has the important consequence that global addresses are unnecessary. The machine ensures that the graph nodes a piece of code is capable of directly interacting with are *already* nearby, and so only a small addressing radius is really needed. Thus each core only needs to be able to address instruction slots on a small neighbourhood of cores. (For dense graphs, intermediate nodes called *forwarding pointers* allow edges to span longer distances than the hardware limit). Intuitively, the machine is taking advantage of the fact that memory accesses are very rarely *actually* random: when traversing a linked list or tree there is a predictable locality pattern that von Neumann machines are blind to because their memory model hides it. This principle is explained in detail in [12].

4 Discussion

The rather different computation world of the SMDG has a number of implications which we will discuss here.

4.1 Advantages

The SMDG makes no distinction between intra-core parallelism and inter-core parallelism, as it has no notion of threads. We think this allows programs to be written with much less concern for the type and quantity of processors available and allows instruction-level parallelism to be spread across multiple cores rather than being trapped within a single oversize and energy-hungry core. The latter idea has been demonstrated for traditional programming models in the Raw microprocessor project [10].

It was mentioned in Section 2 that the SMDG could support safe, efficient inter-process communication. This sounds ideal for microkernel-style component systems whose benefits in terms of software engineering and reliability have long been argued.

We also believe that the SMDG will be particularly useful for applications with irregular parallelism. Such applications are becoming increasingly important and the research community is starting to look for general ways of tackling these kinds of problems [13]. The applications often involve sparse graphical data structures, and we believe that SMDG's ability to exploit locality, even when dependencies are irregular and can change unpredictably at runtime, could be a key strength here.

4.2 Contrasts

The SMDG has some attractive features, and we think it may naturally lend itself to address needs which von Neumann machines address with various ad-hoc mechanisms. The unification of instruction- and thread-level parallelism was mentioned in Section 2. Instead of traps, virtual memory, page tables and memory protection, the SMDG can essentially employ a type-safe instruction set (cf. Singularity [7]).

In the von Neumann model, storage and computation are separate primitives, embodied by the distinction between the left and right hand side of assignment statements [3]. This allows a misbehaving computation to trash any data in its address space, for instance through a buffer overflow. In the SMDG, there is no such dichotomy. Each instruction is an implicit (and indivisible) combination of code and memory, and any combination of instructions is also a combination of code and memory. In pure dataflow programs, the operand slots of each instruction form the memory. In the SMDG, the graph structure also forms part of the memory of a block of code. A misbehaving computation cannot affect any state external to itself except through the capabilities it possesses.

The two typical ways to use a core's local memory are either to make it explicitly programmer-managed, or use it as a cache of global memory (which implies cache coherency protocols etc.). The SMDG enables a different option: the local memory is a node in a self-balancing distributed store. Instead of cache evictions, overfull nodes push data to their neighbours. Further, it is natural to create and manipulate data structures like lists, trees and graphs which span multiple nodes' local memories *without a global namespace*.

4.3 Practicalities

We believe that the SMDG could be a suitable target language for compilers of all sorts of languages, including functional, object-oriented, and data-parallel languages. Most languages discourage or disallow pointer arithmetic, and we think the SMDG can do practically everything else expected of a high-level language. Programmers would not need to write SMDG assembly code, but understanding that the machine is SMDG would influence their code, just as understanding the von Neumann architecture influences the way we code for von Neumann machines. One way to program the SMDG would be to write most components in traditional languages, and use a stitching or orchestration language, for instance StreamIt [17], to wire up these components in parallel.

Experience shows that the von Neumann model, despite its long-identified shortcomings [3], is difficult to dislodge. VLIW machines attempted to change the interface, as did the TRIPS dataflow project [5]. These approaches, however, only targeted instruction-level parallelism (ILP), as they allowed threads to explicitly specify parallel instructions, but were otherwise von Neumann style load-store architectures. Maybe the SMDG has a better chance because it is fundamentally different: it is aimed not at improving single-core performance, but at breaking the barriers between cores and removing the reliance on a global memory.

The SMDG requires certain things to be done in software that are usually done in hardware, and thus could turn into bottlenecks. For example, large arrays can be constructed in software as tree structures, whereas on von Neumann machines they are constructed with pointer arithmetic and `load`/`store` instructions. If beneficial to an application, such data structures can be wrapped in software caches. On von Neumann machines, if the same code is being used on several cores, a read-only copy of the code is cached on all cores. In the SMDG, if multiple copies of code are to be executed in parallel, then multiple copies of the code must be created and can then be spread across the computing resources. Relatedly, function calls must either be inlined or implemented in a kind of client-server style (with the possibility of worker threads being cloned and removed according to demand). The similarities with distributed systems programming are no co-incidence.

Another result of the lack of caching is that unlikely code paths (such as error handlers) are bound to the code, consuming memory and making relocation more expensive, whereas on von Neumann machines these code paths are not cached. Analogous problems affect data structures. There are also many memory, bookkeeping and load balancing overheads that conventional machines do not have. Clearly, these overheads will have a performance impact. It is too early to say whether these overheads are a reasonable cost to pay for making parallelism more accessible.

5 Related Work

A problem affecting many parallel programming models is that they neglect the effect of spatial layout on performance. PRAM is a popular shared-memory model of parallel computation, and it assumes that all memory accesses takes one cycle, which is far from reality. XMT [19] (which is based on PRAM) and BSP [18] both make a distinction between local memory and remote memory, which is an improvement, but this distinction is too coarse for increasing core counts. The von Neumann model simply has (local) registers and (remote) shared memory. GPU programming models add another tier to the middle of the hierarchy: a small block of shared memory which can be shared by a *work group* of threads. Each extra tier places increasing management burdens on the programmer by exposing more of the machine's physical structure. The SMDG does the reverse: it exposes the program's dependency structure to the machine. This makes the machine responsible for managing physical space, yet still gives the programmer much control over spatial optimisation.

The MuNet [20], the $\mathcal{L}$ project [12], and chunk computing [11] all describe similar models with the aim of tying together computing resources. In these models, a large data structure, composed of small chunks of memory which reference each other, stores all (sequential) code and data. All of these projects discuss the importance of improving spatial layout by exploiting the inter-chunk references. The SMDG is similar in motivations and nature, however the chunks are the finest grain possible: individual instructions, and are based on dataflow and graph transformations rather than sequential code. This means compilers do not need to decide how to group instructions into chunks, and it gives the machine more freedom in scheduling and optimisation. A further contribution we make is recognising that type-safe memory and a new instruction set allows us to design with operating systems considerations (such as isolation, virtualisation, and secure communication between mutually untrusted programs) in mind.

Dataflow architectures, particularly [2, 15], have also influenced us. However, we believe that the SMDG has much more in common with the chunk-based models, since these dataflow machines give every word of code and data a global address and allow data structures to be manipulated with loads and stores.

[1] encourages the research community to explore indefinitely scalable hardware architectures. Like ours,

their proposal is also based on the "everything is local" philosophy. They use biologically-inspired "bonds" and "reactions", which we think will be tough to program. We believe that the basic operations of the SMDG, arithmetic, branching and simple graph manipulations, are a better fit to established programming models and the needs of a compiler. The Raw microprocessor [16] is an inherently scalable processor architecture and its motivations and design have influenced our thinking considerably.

6 Conclusion

We have presented the self-modifying dataflow graph (SMDG), not as a means of making better use of parallel computers, but as a way to allow a complete transition from the single-core to the many-core age. The model relieves the programmer of many burdens associated with parallel hardware, without being opaque to reasoning and optimisation. It should permit the use of traditional programming languages. Finally it allows dynamic data structures to be spread across many cores without requiring a global memory, which arguably defeats the von Neumann bottleneck. We cannot yet claim that the model works well in practice, but with ever more parallel hardware looming, we certainly need to be investigating models like this one.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] ACKLEY, D. H., AND CANNON, D. C. Pursue robust indefinite scalability. In *13th HotOS* (Napa, CA, USA, May 2011).

[2] ARVIND, AND NIKHIL, R. S. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comp. 39*, 3 (Mar 1990), 300–318.

[3] BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM 21* (1978), 613–41.

[4] BOKHARI, S. H. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Trans. Softw. Engin. SE-7*, 6 (Nov 1981), 583589.

[5] BURGER, D., KECKLER, S. W., MCKINLEY, K. S., DAHLIN, M., JOHN, L. K., LIN, C., MOORE, C. R., BURRILL, J., MCDONALD, R. G., AND YODER, W. Scaling to the end of silicon with EDGE architectures. *IEEE Comp. 37*, 7 (Jul 2004), 4455.

[6] HILLIS, W. D. *The Connection Machine*. PhD thesis, Massachusetts Institute of Technology, 1988.

[7] HUNT, G., AIKEN, M., FÄHNDRICH, M., HAWBLITZEL, C., HODSON, O., LARUS, J., LEVI, S., STEENSGAARD, B., TARDITI, D., AND WOBBER, T. Sealing OS processes to improve dependability and safety. In *EuroSys* (Lisbon, Apr 2007), pp. 341–354.

[8] KAVI, K. M., BUCKLES, B. P., AND BHAT, U. A formal definition of data flow graph models. *IEEE Trans. Comp. 35* (Nov 1986), 940–948.

[9] LEE, E. A. The problem with threads. *IEEE Comp. 39*, 5 (May 2006), 33–42.

[10] LEE, W., BARUA, R., FRANK, M., SRIKRISHNA, D., BABB, J., SARKAR, V., AND AMARASINGHE, S. Space-time scheduling of instruction-level parallelism on a raw machine. *OSR 32*, 5 (Dec 1998), 46–57.

[11] MAZZOLA PALUSKA, J., PHAM, H., AND WARD, S. Structuring the unstructured middle with chunk computing. In *HotOS* (Napa, CA, USA, May 2011).

[12] MORRISON, J. D. *A scalable multiprocessor architecture using Cartesian Network-Relative Addressing*. MS thesis, MIT, 1989.

[13] PINGALI, K., NGUYEN, D., KULKARNI, M., BURTSCHER, M., HASSAAN, M. A., KALEEM, R., LEE, T.-H., LENHARTH, A., MANEVICH, R., MÉNDEZ-LOJO, M., PROUNTZOS, D., AND SUI, X. The tao of parallelism in algorithms. In *SIGPLAN Notices* (2011), vol. 46, p. 1225.

[14] SWANSON, S., MICHELSON, K., AND OSKIN, M. Configuration by combustion: Online simulated annealing for dynamic hardware configuration. *ASPLOS X Wild and Crazy Idea Session* (Oct 2002).

[15] SWANSON, S., MICHELSON, K., SCHWERIN, A., AND OSKIN, M. WaveScalar. In *MICRO* (2003), p. 291.

[16] TAYLOR, M. B., LEE, W., MILLER, J., WENTZLAFF, D., BRATT, I., GREENWALD, B., HOFFMANN, H., JOHNSON, P., KIM, J., PSOTA, J., SARAF, A., SHNIDMAN, N., STRUMPEN, V., FRANK, M., AMARASINGHE, S., AND AGARWAL, A. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *ISCA* (Munich 2004),, p. 213.

[17] THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. StreamIt: a language for streaming applications. In *Compiler Construction*, R. N. Horspool, Ed., vol. 2304. Springer, 2002, pp. 179–196.

[18] VALIANT, L. G. A bridging model for parallel computation. *CACM 33*, 8 (Aug 1990), 103–111.

[19] VISHKIN, U. Using simple abstraction to reinvent computing for parallelism. *CACM 54*, 1 (Jan 2011), 7585.

[20] WARD, S. A., AND HALSTEAD, JR, R. H. A syntactic theory of message passing. *JACM 27*, 2 (1980), 365383.