Analytical Derivation of Traffic Patterns in Shared Memory Architectures from Task Graphs

Stuart, Matthias Bo; Sparsø, Jens

Published in:
NORCHIP, 2009

Link to article, DOI:
10.1109/NORCHP.2009.5397825

Publication date:
2009

Document Version
Publisher's PDF, also known as Version of record

Analytical Derivation of Traffic Patterns in Shared Memory Architectures from Task Graphs

Matthias Bo Stuart, Jens Sparsø
Department of Informatics and Mathematical Modelling, Technical University of Denmark
Email: {ms,jsp}@imm.dtu.dk

Abstract— **Task Graphs is a commonly used application model in research in computer-aided design tools for design space exploration of embedded systems, including system synthesis, scheduling and application mapping. These design tools need an estimate of the actual communication in the target system caused by the application modelled by the task graph.**

In this paper, we present a method for analytically deriving the worst-case traffic pattern when a task graph is mapped to a multi-processor system-on-chip with a shared memory architecture. We describe the additionally needed information besides the dependencies in the task graph in order to derive the traffic pattern. Finally, we construct a simulator that we use to find the actual traffic pattern in a system and compare this to the derived pattern. Results show that our worst-case derivation overestimates the bandwidth by 9% for systems with small caches and between 32% and 52% for systems with large caches.

I. Introduction

In the development of computer-aided design tools for systems-on-chip (SoCs), abstract models are typically used of the applications running on the SoC and the communication between IP cores (processors, memories, IO controllers, etc.) in the SoC. Two common models are *task graphs (TGs)* and *bandwidth graphs (BGs)*: TGs [1] and variations of these are used to model the dependencies between individual parts – called tasks – of an application, while BGs model the communication between IP cores. Abstract representations of the SoC are also used together with a mapping of the application model (either TG or BG) onto the SoC model.

Often, a simple model for the execution of a TG is assumed that – when combined with a mapping of a TG onto IP cores – results in a one-to-one mapping of the edges in the TG and the edges in the resulting BG. While this correlation may be correct in some cases, it will not be so in general. In this paper, we will explore how to derive a BG from a TG given different memory architectures – focusing on shared memory – in the SoC. We will determine what additional information is required to make an analytical derivation of a BG from a TG and show how to make this derivation.

In order to validate these derivations, we have developed a transaction accurate cache simulator that we use to evaluate the difference between the analytically derived BGs and the actual BGs (the traffic patterns observed in the simulations). This difference shows how much the worst-case derivations overestimate the communication in the system.

Related Work: BGs are typically used in design-space exploration for SoC interconnects, such as Network-on-Chip (NoC) [2]. In [3], a method for synthesizing custom NoC-based interconnects given a BG is presented. In [4], BGs are used for configuring a mixed circuit- and packet-switched interconnect, while in [5], a BG is mapped onto a tile-based SoC.

TGs are also used in a range of situations: In [6], an application model similar to TGs is suggested for benchmarking NoCs, [7] maps a TG onto a SoC with a NoC interconnect, while [8] – besides mapping a TG onto a SoC – also explores the SoC architecture, optimizing it for the given TG. These papers assume a simple correspondence between edges in the TG, the mapping of the TG onto the SoC, and the traffic pattern in the system.

Contributions: The contributions of this paper are 1) a method for analytically deriving worst-case BGs from TGs given two different memory architectures – focusing on shared memory – and 2) an evaluation of the method's accuracy using a simulator.

The rest of this paper is structured as follows: Section II describes TGs and BGs in more detail, section III presents the two memory architectures we consider (local stores and shared memory) and the analytical derivation of a BG from a TG in both architectures, section IV describes our simulator, and section V discusses our results. Finally, section VI concludes the paper.

II. Application Representations

In this paper, we consider two abstract representations of applications: Task Graphs (TG) and Bandwidth Graphs (BG).

A TG is a directed, acyclic graph $\mathrm{TG} = (T, D)$, where a vertex $t \in T$ represents a task, and an edge $d \in D$ represents a (data) dependency of one task on another. An example of a TG is shown in the left of Figure 1.

Vertices, edges, and the graph itself have various properties. Each vertex has a property, l, that describes the task's latency, while each edge has a property, a, that describes the amount of data (number of words) communicated between tasks per execution of the application. The graph has a property, τ, that describes the period of the graph, i.e. how often the modelled application will restart its execution.

As a TG is a model of an application, it also makes sense to consider the "execution" of a TG. A task is ready to execute once all of its predecessors have finished execution. The latency of execution is typically given as a property as described above. The latency of communication is typically

978-1-4244-4311-6/09/$25.00 ©2009 IEEE

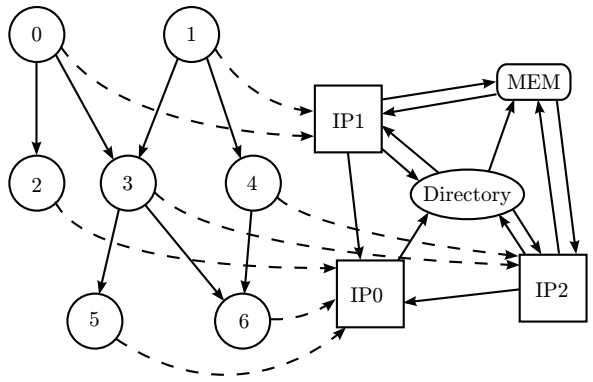

Fig. 1. On the left a TG, on the right a BG. The dashed arrows indicate the mapping of the TG on the SoC.

given as a fixed overhead plus some time per data unit sent. Computation and communication are assumed to be performed disjointly, i.e. once a task starts computing, it will not induce any communication until it has finished computing. These assumptions typically match message-passing communication, where data is *actively sent from* a task, A, *to* another task, B, when A finishes executing. As data transfers in shared memory systems are initiated by the receiving task rather than the sending task, we will assume slightly different execution semantics as described in section IV.

Before describing BGs in more detail, we will first describe the representation of SoCs and mappings of TGs to SoCs. A SoC is represented by a set, I, of IP cores and an interconnect. For the purpose of this paper, we do not need a detailed description of the interconnect. For the set of IP cores, we simply note that it contains one element for each IP core in the SoC, where an IP core represents the circuit that is connected to a terminal on the interconnect. In shared memory systems, I also contains an element for each memory and for each directory. A directory maintains the state of each cache line, which includes granting write permission, issuing invalidations of cache lines, and instructing caches to write cache lines back to memory or sending them to another cache that requests the given cache lines.

A mapping $M : T \rightarrow I$ describes to which IP core a task is mapped – i.e. on which IP core the task executes. After all tasks have been mapped to IP cores, the communication pattern in the SoC can be derived, although in most cases additional information about the SoC memory architecture is needed. This will be discussed in detail in the next section. For now, we define a bandwidth graph (BG), that describes the traffic pattern between IP cores as follows.

A BG is a directed graph $\mathrm{BG} = (I, C)$, where a vertex $i \in I$ represents an IP core in the SoC, and an edge $c \in C$ represents a connection from one IP core to another. Each edge has a property, b, that describes the bandwidth communicated on the connection. Figure 1 shows a TG mapped on a SoC and the BG that results.

III. Memory Architectures

In this paper, we consider two memory architectures: 1) An architecture exclusively using local stores for memory, and 2) a shared memory architecture.

A. Local Stores

In a memory organization using *local stores*, each IP core has some local memory that is explicitly managed by the application, i.e. before data can be accessed, it needs to be explicitly moved to the local store of the IP core to which the task that requires access to the data is mapped. An example of such an architecture is the SPEs of the Cell processor [9].

This architecture matches the typically assumed execution semantics of TGs very well under the assumption that the local stores are sufficiently large to hold the full data set (input and output) of the currently executing task and possibly data from other tasks that are mapped to the same IP core. For some applications, this may be a reasonable assumption, while for others, the quantity of data processed far exceeds the amount of internal memory typically found in a SoC.

The traffic pattern resulting from mapping a TG on a system using local stores matches the dependencies between tasks very well: A dependency from task A to task B with a data amount a results in a connection from $M(A)$ to $M(B)$ with a bandwidth $b = a/\tau$ as long as $M(A) \neq M(B)$, i.e. A and B are executing on different IP cores. Dependencies between tasks executing on the same IP core do not result in connections in the BG.

Depending on the specific implementation of the local store architecture, additional bandwidth and/or connections may be required to handle the communication protocol. For the case of A actively sending its output to $M(B)$'s local store, b will include some overhead to indicate where in the local store to put the data. If DMA transfers are available, this overhead should be minimal. On the other hand, if B needs to actively retrieve its input from $M(A)$'s local store, a connection from $M(B)$ to $M(A)$ will be needed with the bandwidth on this connection equal to the protocol overhead. Finally, a third party, C, may be acting as a controller, thereby requiring bidirectional connections between $M(A)$ and $M(C)$ and between $M(B)$ and $M(C)$, again with the protocol overhead as the bandwidth.

B. Shared Memory

Shared memory architectures are probably the best known of the two architectures considered here. Data is uniquely identified by an address in memory and is accessed only using that address. The memory may be organized in different ways such as distributed on-chip with a portion of memory at each IP core, or off-chip with one or more IP cores being interfaces to the off-chip memory.[1] The methods we present here are independent of the specific memory organization – we only need a map of address ranges to memories. For simplicity,

[1]Distributed off-chip is theoretically also an option, although the number of pins required to support so many memory interfaces makes this option very unrealistic.

we require that each data set (each input/output of a task) is in a contiguous region of memory. The extension to non-contiguous regions is trivial and may in fact be handled using the methods described in this paper simply by using parallel edges in the TG for each contiguous region.

At first, we will consider a system without caches. Although this case is of limited practical interest, it is useful to consider before including the effects of caches. When caches are not present, all accesses to data must go directly to main memory. A task therefore needs bidirectional channels to the memory or memories containing its input data (a channel *to* the memory for requests and a channel *from* the memory for data), while only a unidirectional channel is needed to the memory/memories containing the output data as this memory is only written to, not read from. The additionally needed information for deriving a BG from a TG in a shared memory system *without* caches is thus only the location of data and a map of addresses to memories.

Adding caches to the system introduces a few additional aspects to consider when deriving a BG from a TG. The data at a given address may be located at several places in the SoC at the same time: Both in the portion of main memory that contains the given address and in one or more caches. Furthermore, the value of the data may differ between these locations, requiring the system to figure out which value is correct when an IP core requests the current value (or rather to invalidate the incorrect versions when the value is updated). How to accomplish this is an extensively researched subject [10] which we will not go into details with here.

We assume that a task only reads from its input addresses and writes to all of its output addresses. Additionally, for the SoC, we need to consider the cache organization, which we assume to be fully associative with LRU replacement policy.

For maintaining cache coherence, we make use of cache directories. We will not go into details with the protocol here, but simply list the possible transactions. An IP core sends a message to the directory when it needs a cache line (either for reading or writing) that it does not already have and when it evicts a cache line. When the directory receives a request for a cache line, it checks the state of that line. The possible states are *uncached*, meaning that the cache line is only in main memory, *shared*, meaning that the cache line can be found in one or more caches, or *modified*, meaning that the cache line can be found in exactly one cache, where its value has been updated. Depending on the state, the directory sends a message to either main memory, one of the sharers, or the cache with the modified value, and instructs that one to send the cache line to the requesting IP core. If the request is a write, the directory also sends messages to all other caches containing the given line telling them to invalidate the line.

In our derivation of a BG from a TG, we find the worst-case traffic pattern, i.e. if a cache line can potentially be found in multiple places, we will add bandwidth to the traffic pattern corresponding to retrieving the cache line from *all* these places. We also assume that the TGs are free from race conditions, i.e. if multiple tasks write to the same address,

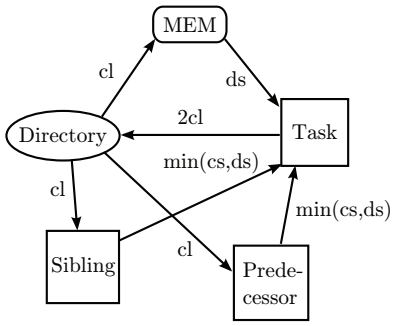

Fig. 2. The contribution to a BG from a read.

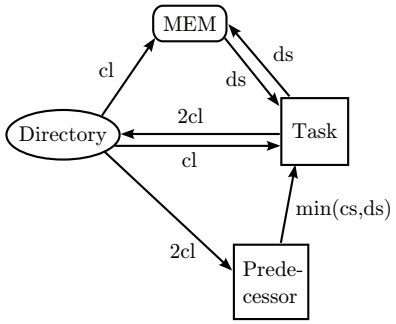

Fig. 3. The contribution to a BG from a write.

a dependency exists between these tasks forcing a unique ordering of the tasks. Finally, we do not consider the use of burst transfers currently, meaning that our protocol overhead will be significantly larger than in a system with this capability. In the following, we go through the process of analytically deriving a BG from a TG given the above assumptions and a mapping of the TG on the SoC.

Each edge in a TG represents data being first written and then read. We will separate the contributions of an edge to the BG in these two cases. When data is read, it results in the worst-case BG contribution shown in Figure 2, where cl is the number of cache lines, ds is the size of the input in words, and cs is the size of the cache also in words. Note that the "sibling" node means any sibling in the TG that has the same input. From the task, up to two times the number of cache lines messages are sent to the directory: One for requesting the cache line, and one for notifying the directory that the line has been evicted. The data may be found in either memory, in the predecessor's cache, or possibly in the cache of a sibling in the task graph. Thus, the directory sends cl messages to each of these. The memory may reply with the whole data set, while the predecessor and the sibling may reply also with the data set but at most the size of their respective caches.

When data is written, the resulting BG contribution is as shown in Figure 3. First, notice the absence of the sibling: This is due to our requirement that race conditions do not occur, meaning that if a task writes an address, its sibling cannot have read it, as this would impose a specific ordering on the execution of the tasks without an explicit dependency. The situation where a task is both sibling and predecessor is

handled through its predecessor relation. Second, notice that multiple parts of the graph is identical to that for a read. This is because the cache lines must be fetched even on a write, as there in general is no guarantee that the entire cache line will be written to. The additional contribution to the BG compared to the read-case are cl messages from the directory to the task informing it that the cache line's state has been changed to shared, and from the directory to the predecessor (if any) that last wrote the given addresses instructing it to invalidate its cache entries. Finally, the written values will need to be written to main memory as well at one point meaning the transfer of ds data words from the task to main memory.

A BG is derived from a TG by applying these derivations across the entire TG. In order to get actual bandwidths instead of data amounts as described above, it is necessary to divide by the period of the TG.

IV. Simulator

We have implemented a simple cache simulator that executes a TG in order to compare our derived BG to an actual one. We have chosen execution semantics for the TG that more closely resemble a shared memory system. Specifically, communication is part of the task's execution in our simulator. A task reads each input address once and writes each output address once. When reading and writing, a model of a cache is used to check if the given address is already cached and if not, to send a message to the directory. The cache coherence protocol follows the description in the previous section. By counting the number of messages and cache lines sent in the interconnect, we can construct an actual BG that we use to compared with the analytically derived BG.

In order to simulate a TG, a mapping of tasks to IP cores is needed together with a schedule. As the focus of this paper is not on optimizing either, we simply use a random mapping with ASAP scheduling on each IP core. In the experiments, the mapping for a given TG on a given SoC is the same for both the analytical derivation and for the simulation.

V. Results

In this section, we present our experiments and results.

We have generated 16 different TGs, four each with 16, 32, 128, and 1024 tasks. Additionally, we have generated four different SoCs by varying two parameters over two values each: 16 or 64 IP cores, and cache sizes of 1 or 256 cache lines. For each TG, we have generated 5 different random mappings to IP cores and a schedule for each of these mappings. This results in 320 combinations of TGs, SoCs and mappings/schedules, for each of which we have generated BGs through both analytical derivation and simulation.

The first observation about our results is that the mapping/schedule has negligible impact on the deviation between our worst-case analysis and the simulation results. The second observation is that the number of tasks in the TG only has a moderate impact on the accuracy of our analytical derivation: The percentage overestimation of our worst-case analysis varies between 40% and 52% for TGs with 16 tasks and between 32% and 35% for TGs with 1024 tasks in the SoCs with large caches. The difference in accuracy is explained by the fact that for small TGs, much less data will be evicted from caches leading to significant overestimation of the bandwidth between memory and caches. However, for small caches, the overestimation is independent of the task size and drops to around 9%. This difference is explained by the inherent difficulty in estimating the impact of caches: When caches are small, our worst-case analysis correctly estimates that most of the data comes from main memory, while very little is transferred directly between caches, while when caches are large, our analysis estimates the full data set being transferred both from main memory *and* directly between caches, while in reality, it will only be transferred from one of the two.

VI. Conclusion

In this paper, we have presented a method for analytically deriving a bandwidth graph from a task graph. The analysis is performed using worst-case estimates of the required communication in a shared memory system and takes both caches and the cache coherence protocol into consideration.

By comparing our analytically derived bandwidth graphs with graphs generated by a simulator, we observe that for a system with very small caches, our estimates are within 9% of the actually measured traffic, while for larger caches, the accuracy varies between 32% and 52%, depending on the size of the task graph.

References

[1] R. Dick, D. Rhodes, and W. Wolf, "Tgff: task graphs for free," *Hardware/Software Codesign, 1998. (CODES/CASHE '98) Proceedings of the Sixth International Workshop on*, pp. 97 –101, 1998.

[2] W. J. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks," in *Proc. of the 38th Design Automation Conference (DAC)*, June 2001. [Online]. Available: http://citeseer.ist.psu.edu/dally01route.html

[3] J. Chan and S. Parameswaran, "NoCOUT : NoC topology generation with mixed packet-switched and point-to-point networks," *13th Asia and South Pacific Design Automation Conference ASP-DAC 2008*, pp. 265–270, 2008.

[4] M. B. Stuart, M. B. Stensgaard, and J. Sparsø, "Synthesis of Topology Configurations and Deadlock Free Routing Algorithms for ReNoC-based Systems-on-Chip," *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2009*, 2009.

[5] J. Hu and R. Marculescu, "Energy- and performance-aware mapping for regular NoC architectures," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 4, Apr. 2005. [Online]. Available: http://www.ece.cmu.edu/šld/pubs/papers/T-CAD_April05.pdf

[6] E. Salminen, C. Grecu, T. D. Hämäläinen, and A. Ivanov, "Network-on-chip benchmark specification part 1: Application modelling and hardware description version 1.0," OCP (http://www.ocpip.org), Tech. Rep., 2008.

[7] T. Lei and S. Kumar, "A two-step genetic algorithm for mapping task graphs to a network on chip architecture," *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, pp. 180–187, 2003.

[8] J. Madsen, T. K. Stidsen, P. Kjærulf, and S. Mahadevan, "Multi-objective design space exploration of embedded system platfoms," in *IFIP Working Conference on Distributed and Parallel Embedded Systems*. IFIP, oct 2006, intivted paper. [Online]. Available: http://www2.imm.dtu.dk/pubdb/p.php?4574

[9] M. Kistler, M. Perrone, and F. Petrini, "Cell multiprocessor communication network: Built for speed," *IEEE Micro*, vol. 26, no. 3, pp. 10–23, 2006.

[10] P. Stenström, "A survey of cache coherence schemes for multiprocessors," *Computer*, vol. 23, no. 6, pp. 12–24, June 1990.