Impact of PCI-Bus Load on Applications in a PC Architecture

Sebastian Schönberg
Operating Systems Research Group
University of Technology Dresden, Germany*
schoenbg@os.inf.tu-dresden.de

Abstract

Any data exchanged between the processor and main memory uses the memory bus, sharing it with data exchanged between I/O devices and main memory. If the processor and a device try to transfer data at the same time, an impact can be seen on the processor as well as on the device. As a result, the execution time of an application on the processor may increase due to the memory-bus load generated by I/O devices. In real-time environments, this impact can result in missed deadlines and a behavior that is different to that intended by the designer of the system.

This paper gives a method for describing and quantifying the impact of such load on applications executed by the processor.

1. Motivation

In classical hard real-time environments, buses have never been a problematic resource. To guarantee bandwidth and latency, either dedicated (bus) hardware with real-time support is used or the available bus bandwidth is higher than the bandwidth required by applications. More recently, an increasing number of applications processing a high volume of data such as audio and video have appeared. These modern applications would run well in classical real-time environments, however, they are targeted to run on standard commodity PC hardware. A problem arises since PC hardware is less predictable than dedicated hardware and does not provide support for bus-bandwidth reservation.

In PC systems, processors and the main memory are tightly coupled via the *memory bus*, sometimes also referred to as *local bus*. To prevent the memory bus from becoming the critical performance bottleneck of the system, this bus should and usually does provide the highest bandwidth in the system. Memory-bus bandwidth is shared among all processors and devices accessing main memory. If the system does not provide mechanisms to assign a certain bandwidth to a device, the device's bandwidth always depends on the actual behavior of all other devices. Since this also holds for processors, one can conclude that the execution times of processor instructions accessing main memory also depend on the behavior of other devices. Hence, the execution time of an entire application can vary and is in¤uenced by other devices.

Early measurements showed that the impact on the execution time of applications is relevant and cannot be neglected when large amounts of data are transferred [16]. Hence, the na˝ve assumption that the execution time of an application is not in¤uenced by other devices (as often seen in real-time systems) may result in violations of the given guarantees. To deal with that effect, resource reservations based on the execution time of an application must be adapted to handle the impact caused by bus traf£c.

In this paper we describe a method for quantifying the impact of load generated by PCI-devices on the execution time of applications.

2. Overview

To characterize bus loads on the system, we follow the terminology of Pentium processor's performance-counter speci£cation, the *internal load* of a speci£c processor refers to the memory-bus load generated by this processor, and the *external load* refers to the memory-bus load generated by all other possible sources. In the uniprocessor case, external load is always generated by the PCI host bridge. An *internal transaction* of a speci£c processor refers to a transaction on the memory bus generated by this processor, and an *external transaction* refers to a memory-bus transaction from any other possible sources, respectively. In the uniprocessor case all external transactions are generated by the PCI host bridge.

Since external load reduces the capacity of the memory bus available to the processor, we can assume also that it increase the execution time of an application. In real-time systems, the correct amount of processor resource must be reserved to successfully execute an application. To consider the impact of external bus load on applications requiring resource reservations, we need a metric to express the strength of the impact. We call the ratio between the execution time

* This work has been supported by Intel Corporation

under external load and the execution time under no external load the *slowdown factor*.

The slowdown factor is always greater than or equal to one. The slowdown factor of an application A under external load L is denoted by $\mathcal{F}(A, L)$. Depending whether the application A or the load L remain £xed, the notation can be abbreviated by $\mathcal{F}_A(L)$ or $\mathcal{F}^L(A)$. The external load can be expressed by the bandwidth or the number of memory-bus transactions per time unit.

For reservation schemes that consider worst-case execution times—as for hard real-time systems—the worst-case impact is of interest. To determine this worst-case impact, all possible factors such as buffer and queue states in the host bridge, the access pattern of the physical RAM chips, and processor features such as the Pentium's system management mode[1] must be considered in addition to the application itself. A detailed analysis here is beyond the scope of this paper. For simplicity, we consider increasing the external load to a maximum possible value as an acceptable approximation of the worst case: $\mathcal{F}(A, L \rightarrow \max) = \mathcal{F}^{L_{max}}(A)$. We call this value the *worst-case slowdown factor* of an application (WCSF).

Alternatively to the worst-case load, we can identify an application that is most sensitive to external load (worst-case application). In this case, we denote $\mathcal{F}(A \rightarrow \max, L) = \mathcal{F}_{A_{max}}(L)$. The combination of worst-case load and worst-case application leads to an *upper-bound worst-case slowdown factor* for a machine. We describe this value by $\mathcal{F}(A \rightarrow \max, L \rightarrow \max) = \mathcal{F}^{L_{max}}_{A_{max}}$. To determine the con£dence level of these worst-case approximations, analytical and statistical techniques such as those proposed by Edgar *et al.* [7] can be used.

The slowdown factor and the worst-case slowdown factor are metrics that describe the strength of the impact caused by external bus load. For example, an application A characterized by $\mathcal{F}^{L_{max}}(A) = 1.8$ takes, under worst-case load, 80 percent more time to complete. However, $\mathcal{F}(A, 20\text{MB/s}) = 1.2$ means that the application runs 20 percent longer under the given load of 20MB/s. The upper-bound worst-case slowdown factor gives a maximum value by which the most sensitive application can be in¤uenced.

In the following sections, we describe three approaches for obtaining the different slowdown factors of an application.

3. Empirical Approach

The £rst approach to determining the WCSF of an application is purely empirical. In uniprocessor systems, the PCI bus is the only source for external load not generated by the processor(AGP cards can be considered as a PCI-bus extension). We obtain the WCSF by saturating the PCI bus and measuring the execution time of the application in contrast to the application's execution time under no external load. To generate such PCI-bus load, we used £ve identical bus-master-capable FORE PCA200e ATM network cards, each with a programmable PCI-bus interface.[2] The processor on the network card was used to initiate PCI-bus read or write transactions of variable burst length. We achieved a maximum write data transfer rate of up to 118MB/s, which is also given by the manual as the maximum value of the host bridge [10].

To obtain some representative results, we chose three different applications: The data encryption standard (DES), for an application with low internal load; a sorting algorithm (Quicksort), for an application with medium internal load; and raw-data transfers, for an application with high internal load. The classi£cation into low, medium and high internal load is based on the number of CPU cycles required for precessing one 32-bit word as shown in Table 2. All applications were implemented to process data (in 32-bit words) from a 2MB source buffer and write the results to an adjacent 2MB target buffer. They were designed to touch source and target buffers exactly once per 32-bit word

We disabled any caching for the source and target-buffer region but not for any other memory area.[3] As a result, any data word of the source buffer must be individually fetched from main memory and any data word written to the target buffer is forced to main memory. This makes the access to the source and target buffer predictable. The same effect can be achieved by processing only one 32-bit word out of every cache line with a read-allocated, write-through cache. To determine the impact of external load in conjunction with caching, we performed the same tests with caching enabled for both source and target buffer.

The number of transactions on the memory bus was measured using the processor's internal performance counters. IA-32 processors provide performance counters for internally generated (t_{int}) and all (t_{tot}) memory-bus transactions but not for externally generated memory-bus transactions (t_{ext}). On a uniprocessor system, where only one processor and the host bridge can generate memory bus load, the number of external transaction is:

$$t_{ext} = t_{tot} - t_{int}. \tag{1}$$

Characterizing the type of external transaction leads to the distinction between external read transactions (t^r_{ext}) and external write transactions (t^w_{ext}).

1 For instance, Pentium's system management mode (SMM) is used for the emulation of legacy devices or to work around bugs of the motherboard or in the chipset.

2 The ATM card uses a 50MHz embedded version of the Intel i960 processor, 2MB application memory, and a proprietary PCI-bus interface [5].

3 Caching policies can be de£ned for "power of 2" multiples of 4KB memory frames ($2^{(2+n)}$KB) aligned to their size in the physical address space by use of the processor's memory-type range registers (MTRRs).

3.1. DES Algorithm

DES [19] is a well-known symmetric cryptographic system. It uses a 56-bit key to encrypt and decrypt data. In sixteen iteration steps, the input data is permuted and written to the target buffer. The C-code implementation of this permutation uses local variables to store temporary results. Since IA-32 processors have a small number of available registers, these variables are stored on the stack frame of the procedure in memory. Recall that only source and target-buffer accesses are uncached; stack and local-data accesses are cached. Due to the large number of temporary local variables, DES has strong data locality, which leads to a high cache-hit rate. It is very likely that temporary results are held completely in the £rst- or second-level caches and are never actually written to main memory. We measured an $\mathcal{F}^{L_{max}} = 1.05$ for the encode and $\mathcal{F}^{L_{max}} = 1.03$ for the decode operation.

With caching enabled for the buffer, the application was about 26% faster and generated only 9% of the memory-bus load (430,000 versus 4,560,000 transactions per second). However, in relation to the 265 million external transactions generated by the PCI cards at the same time, the increase from 430,000 to 4,560,000 internal transactions (0.16% to 1.7% in relation to the number of total transactions) is negligible and the $\mathcal{F}^{L_{max}}$ values are identical.

3.2. Sorting algorithm (Quicksort)

Quicksort is a fast sorting algorithm based on divide-and-conquer. An array is repeatedly divided into two halves until two adjacent items can be directly compared against each other, and exchanged if necessary. We implemented an iterative version described by Sedgewick [17, 18] that sorts the 2MB source buffer into the 2MB target buffer. The measured worst-case slowdown factor without caching is $\mathcal{F}^{L_{max}} = 1.09$.

3.3. Raw-Data Transfer

In the two applications described above, memory accesses were only a small or medium fraction of all executed instructions. Under these circumstance, the WCSF values range between 3% and 10%.[4]

Figure 1 shows the maximally possible number of CPU-generated internal transactions in relation to the number of external transactions. Figure 2 shows the derived slowdown factors. Each graph in both diagrams represents a measurement with a different combination of external and internal transaction types, *e.g.*, a combination of read and write transactions. For example, read–write means that the PCI-bus devices execute main-memory read operations while the processor executes main-memory write operations. Table 1 summarizes the worst-case slowdown factors.

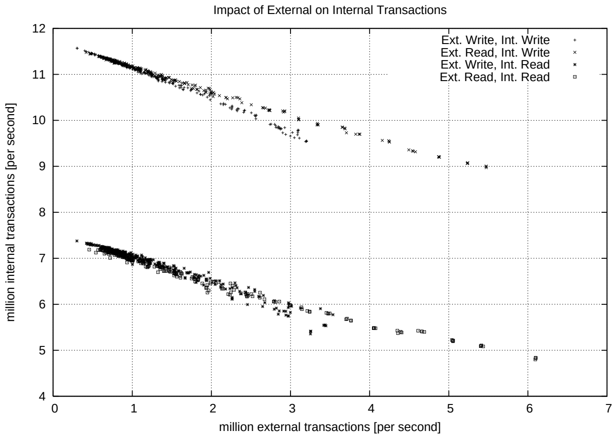


Figure 1. Impact of external transactions on processor-generated (internal) memory transactions

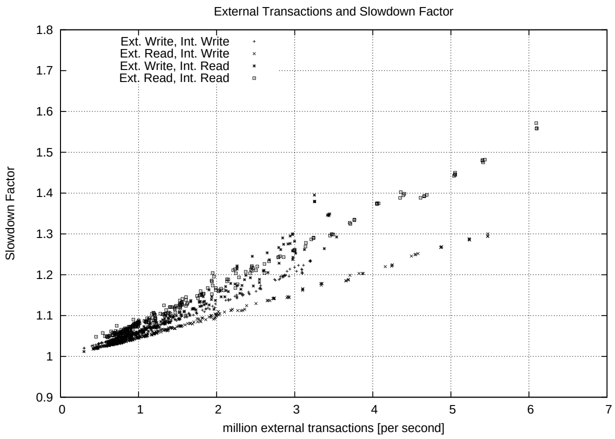


Figure 2. Resulting slowdown factors, calculated from Figure 1

3.4. Summarizing Measurement Results

Table 2 gives a detailed overview of the number of processor cycles, the number of transactions on the memory

4 Earlier measurements [16] on an older machine (Pentium 90MHz) showed slightly higher slowdown factors. DES: $\mathcal{F}^{L_{max}}$=1.03; Quicksort: $\mathcal{F}^{L_{max}}$=1.1. Additionally, we measured decoding an "IPP-coded" MPEG-1 video (384x288 pixel, 24 bit color, compression factor 1:60 for 33 I-frames, 1:169 for 66 P-frames) with $\mathcal{F}^{L_{max}}$=1.36 and data transfer: read $\mathcal{F}^{L_{max}}$=2.15, write $\mathcal{F}^{L_{max}}$=1.18, copy $\mathcal{F}^{L_{max}}$=1.21.

Operation	CPU Cycles per 32-bit word	Bus transactions per 32-bit word	$\mathcal{F}^{L_{max}}$
memory read	55.5	1	1.49
memory write	35.1	1	1.26
memory copy	99.4	2	1.35
Quicksort	457.7	2	1.09
DES encode	701.3	2	1.05
DES decode	699.2	2	1.03

Table 2. CPU cycles to process one 32-bit input word, bus transactions per 32-bit word

$\mathcal{F}^{L_{max}}$	CPU read	CPU write
PCI read	1.49	1.26
PCI write	1.38	1.21

Table 1. WCSF for processor-issued memory operations, derived from Figures 1 and 2

bus to process one 32-bit word, and the resulting worst-case slowdown factors.

Obviously, code executed on the host processor that performs only memory read operations to generate internal memory-bus load is the most sensitive to write operations as external bus load. We consider the slowdown of this application accessing as the upper-bound WCSF ($\mathcal{F}^{L_{max}}_{A_{max}}$) for this system. For our machine, we determined an upper-bound value of 1.49.

Knowledge of this value allows for a very simple test for admission that considers the impact of PCI-bus load. If an application can be scheduled even with the upper-bound WCSF, the application can be scheduled under any load possible on that machine. If it cannot be scheduled under the upper-bound WCSF but under no external load, the application-speci£c WCSF must be considered.

4. Algorithmic Approach

The advantage of an empirical approach is its simplicity—once the measurement environment has been built, only a simple test must be performed. However, the WCSF depends on many factors of the system used and individual measurements must be taken on every system.

For schemes based on the slowdown factor, not only one but a series of measurements under various external-load values must be taken. This makes the empirical approach expensive. To overcome this disadvantage, we consider an algorithmic approach and strive to calculate the slowdown factor based on characteristic values of both the algorithm and the hardware.

4.1. Calculation of Application Worst-Case Slowdown Factor

In the previous sections, we showed that only memory operations are affected by external bus load, and we have described the maximally possible in¤uence by the upper-bound WCSF on the execution time of applications. In this section, we strive to determine an application's worst-case slowdown factor as a combination of separate characteristics of the application and of the machine. The major advantage of such a separation is that an application can be described independent of the underlying machine. In combination with the characteristics for the machine, the machine-speci£c slowdown factor is determined. However, it will become clear later that we can achieve this separation only with certain limits.

From related work described in Section 5.2, we know that the execution time of an application is the linear combination of the number of times each abstract operation of the instruction mix is executed (C_i), multiplied by the time it takes to execute each operation (P_i). The execution time (T) of an application using an instruction mix with n individual abstract operation can be determined by the following:

$$T = \sum_{i=1}^{n} C_i P_i. \quad (2)$$

In terms of bus-induced impacts on applications, it is suf£cient to describe an application by an instruction mix that divides the operations into three abstract operations: memory read-sensitive (C_r), memory write-sensitive (C_w) and non-memory-sensitive (C_o) operations. Given the total number of executed operations by $C_{tot} = C_w + C_r + C_o$, we can extend Equation 2 to:

$$\begin{aligned} T &= C_r P_r + C_w P_w + C_o P_o \\ &= C_{tot}\Big(\frac{C_r}{C_{tot}} P_r + \frac{C_w}{C_{tot}} P_w + \frac{C_o}{C_{tot}} P_o\Big). \end{aligned} \quad (3)$$

The time for an operation (P) is determined by the CPU frequency (f) and the number of CPU cycles this instruction takes (cyc). We can substitute P_x with $\frac{cyc_x}{f}$, and rewrite $\frac{C_x}{C_{tot}}$ as S_x, the share each instruction has on the amount of total instructions, and obtain:

$$T = \frac{C_{tot}}{f}\Big(S_r cyc_r + S_w cyc_w + S_o cyc_o\Big). \tag{4}$$

A coarse estimation of an application's WCSF is based on the assumption that read and write instructions are slowed down by the upper-bound WCSF ($\mathcal{F}_{A_{max}}^{L_{max}}$). The application under maximum external load is now executed in the time $T^{L_{max}}$:

$$T^{L_{max}} = \frac{C_{tot}}{f}\Big(S_r cyc_r \mathcal{F}_{A_{max}}^{L_{max}} + S_w cyc_w \mathcal{F}_{A_{max}}^{L_{max}} + S_o cyc_o\Big). \tag{5}$$

The WCSF for the application can be determined from Equations 4 and 5:

$$\begin{aligned} \mathcal{F}_{A}^{L_{max}} &= \frac{T^{L_{max}}}{T} && (6)\\ &= \frac{S_r cyc_r \mathcal{F}_{A_{max}}^{L_{max}} + S_w cyc_w \mathcal{F}_{A_{max}}^{L_{max}} + S_o cyc_o}{S_r cyc_r + S_w cyc_w + S_o cyc_o} && (7) \end{aligned}$$

We can infer from Equation 6 that it is suf£cient to describe the instruction mix as a triple IM that contains the shares of read, write, and all other operations. All other values are machine dependent and can be denoted by a machine descriptor MD:

$$\mathrm{IM} = (S_r, S_w, S_o) \quad \mathrm{MD} = (cyc_r, cyc_w, cyc_o). \tag{8}$$

Ideally, the parameters of the machine descriptor depend only on the structure and capabilities of the processor. However, due to performance-improving features such as pipelining and parallel execution of instructions, which make the processor less predictable, these values can vary and are also in¤uenced by the application.

In a more accurate formula, we consider the difference between read worst-case slowdown factor ($\mathcal{F}_{r}^{L_{max}}$) and write worst-case slowdown factor ($\mathcal{F}_{w}^{L_{max}}$). Both values can be taken from measurements of the system. For our measurement system, they are shown in Table 2. The modi£ed formula can be derived from Equation 6:

$$\mathcal{F}_{A}^{L_{max}} = \frac{S_r cyc_r \mathcal{F}_{r}^{L_{max}} + S_w cyc_w \mathcal{F}_{w}^{L_{max}} + S_o cyc_o}{S_r cyc_r + S_w cyc_w + S_o cyc_o}. \tag{9}$$

To verify these results, we applied the measurement results of simple read and write operations ($cyc_r = 55.5, cyc_w = 35.1$) as given in Table 2 to calculate the worst-case slowdown factor of the memory-copy application. The source code in assembly language of the memory-copy application is shown in Figure 3. The integer operations of lines 2, and 4, and the branch operation of line 6 are paired by the processor with the instructions in lines 1, 3, and 5, respectively. Since the code allows pairing of all instructions, the cycle value of integer operations (cyc_o) is 0.5. To avoid in¤uence by the number and order by which memory modules are plugged into the system, the code is written so that no two 32-bit words can be merged into one single 64-bit memory-bus transaction. The same holds for the read and the write test.

```
(1) l1: mov   eax, dword ptr [esi]
(2)     add   esi, 8
(3)     mov   dword ptr [edi], eax
(4)     add   edi, 8
(5)     dec   ecx
(6)     jnz   l1
```

Figure 3. Machine code in assembly language for "copy" application

The shares for read and write are each $1/6$, the share for other (non-memory-accessing) operations is $4/6$. Applying these values to Equation 6 results in a calculated slowdown factor $\mathcal{F}^{L_{max}} = 1.392$. This compares favorably to the measured slowdown factor for the memory-copy application of 1.35.

We performed the same calculation for the DES application. In this case, instructions cannot be paired so nicely, since we have many data dependencies. We have measured an average value of 0.9 cycle for executing a non-memory operation. We have seen one memory read and one memory write bus request per 750 executed instructions. These values are identical for encode and decode operations. Applying the previously given equations results in the parameters given in 10.

$$\begin{aligned} \mathrm{IM} &= \Big(\frac{1}{750}, \frac{1}{750}, 1 - \frac{2}{750}\Big) && (10)\\ \mathrm{MD} &= \Big(55.5, 35.1, 0.9\Big) &&\\ \mathcal{F}^{L_{max}} &= 1.047. && (11) \end{aligned}$$

This measured slowdown factor is $\mathcal{F}^{L_{max}} = 1.05$ for DES encode and $\mathcal{F}^{L_{max}} = 1.03$ for DES decode. These results show that the proposed solution is also applicable to relatively complex operations and not only to trivial operations such as memory copy.

4.2. Calculation of Application Slowdown Factor

All previous considerations were based on the worst-case slowdown factor for combinations of read and write operations, or in the worst case on the upper-bound WCSF of the system. As common to all worst-case-based reservation schemes, they also lead to an over-reservation and a waste of resources.

If reservations can be based on the real PCI-bus load, available resources can be utilized better. To determine the PCI-bus load in advance, all devices must announce their future bandwidth consumption at a central instance of the operating system. With cooperating resource managers, DROPS [8, 3] already provides an applicable scheme where

PCI-bus bandwidth reservations can be made. Legacy device drivers, which cannot give exact information about the generated PCI-bus load, can give worst-case approximations based on information about the card. The PCI-bus interface chip used often provides information adequate to determine the maximum possible PCI-bus load of a card. Even more trivial, the type of the card can be considered. A 100Mbit network card is barely capable of generating more than 12.5MB/s of sustainable PCI-bus load. If the driver detects that the card is only connected to a 10Mbit network, it is suf£cient to assume a maximum bandwidth of 1.25MB/s.

In the next step, we determine the slowdown factor in relation to a given bus load. All previous considerations were based on transactions. Hence, we have to convert a load given in MB/s into a load given in memory-bus transactions per second. If caching is enabled, the processor always reads and writes an entire cache line, instead of each individual memory cell and we see only one bus transaction per cache line. Hence, the number of CPU-generated memory-bus transactions does not depend only on the number of memory instructions, but also on the caching behavior of the application. The number of bus transactions caused by a certain PCI-bus load is easier to determine. The memory-bus interface of the host bridge has only a few read-buffer or write-buffer entries. Since all relevant transfers to or from main memory are bursts, the bridge combines multiple data words into one single memory-bus transaction. The result is an almost linear relation between PCI-bus bandwidth and the number of transactions on the memory bus.

The in¤uence of external transactions on the maximal possible number of internal transactions on accessing memory has already been shown in Figure 1. We see one transaction on the memory bus for a 32-byte write access or a 16-byte read access. These two values are host-bridge dependent. In the test-bed with £ve PCI cards, the maximum achieved PCI-bus bandwidth is about 102MB/s (6.1M transactions) for read and 118MB/s (3.5M transactions) for write operations.

The calculation of the slowdown factor is based on Equation 6. We replace the upper-bound worst-case slowdown factor by the slowdown factors for read and write under a certain load. To determine $\mathcal{F}_r(t_{ext})$ and $\mathcal{F}_w(t_{ext})$ from the number of external transactions, we consider two methods. The simpler method to get the slowdown factor uses a lookup-table £lled with slowdown-factor values taken from the tests described in Sub-section 3.3. If an exact value is not available, an approximated value is used. A drawback is the required amount of data that must be available for the table.

A more elegant version approximates the slowdown factor by a monotonically increasing function. At a glance, one might expect a linear dependency between the slowdown factor of an application and the external bus load. However, analyzing Figure 2 shows that a polynomial function of order two (k=2) (*i.e.*, a quadratic function) approximates the relation more accurately. The reason for a non-linear dependency is that contention on the bus increases with the square of the bus utilization. If memory-bus speed is high and the length of each memory-bus transaction is short, contention reduces and the resulting almost linear relation can be described with a β_2 coef£cient of zero.

The quadratic function of order two is given by $f(x) = \beta_2 x^2 + \beta_1 x + \beta_0$, with x as the number of external transactions per second. To calculate the actual value for $\mathcal{F}_A(t_{ext})$, we obtain:

$$\mathcal{F}_A(t_{ext}) = \beta_2 \Big(t_{ext}\Big)^2 + \beta_1 \Big(t_{ext}\Big) + \beta_0. \tag{12}$$

The β coef£cients are machine dependent and must be determined for each machine individually. In order to £nd the best approximation, we consider the biased variance σ^2 of the pairs (x_i, y_i) of all measured samples:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}\Big(y_i - (\beta_2 x_i^2 + \beta_1 x_i + \beta_0)\Big)^2 \tag{13}$$

must be minimized using the non-linear *least-squares-£tting method*. We can derive the Equation system 14 for the polynomial function of order two:[5]

$$\begin{aligned} \beta_0[x^0] + \beta_1[x^1] + \beta_2[x^2] &= [y] \\ \beta_0[x^1] + \beta_1[x^2] + \beta_2[x^3] &= [yx] \\ \beta_0[x^2] + \beta_1[x^3] + \beta_2[x^4] &= [yx^2]. \end{aligned} \tag{14}$$

Solving the Equation system 14 with our measured values leads to the tuples of coef£cients as given in Table 3. Since the slowdown factor of an application running on a system without external bus load ($\mathcal{F}(0)$) is one, the β_0-coef£cient must be 1 as well. This holds for all our calculated coef£cients.

Figure 4 shows measured values identical to Figure 2 but also shows the approximated polynomial functions.[6] One external write transaction transfers 32 bytes from the host bridge to main memory, but one external read transaction transfers only 16 bytes. Hence, the impact on applications of write transactions is increases quicker than the impact of read transactions.

Figure 5 shows that the weighted absolute errors e_i, as given in Equation 15, are less than 6%. This indicates that a quadratic function approximates the behavior of the real machine very well.

$$e_i = \left|\frac{y_i - f(x)}{f(x)}\right|. \tag{15}$$

Another parameter that describes the exactness of an approximated function is the mean error σ_y which can be cal-

5 Notation as de£ned by Gauss: $[x] = \sum_{i=1}^{n}(x_i)$.

6 For a better visibility, we have split up Figure 4 into four graphs shown on Page 12. The original color graphs are available on request.

$(\beta_2, \beta_1, \beta_0)$	**CPU operation**	
	read	**write**
PCI read	(0.7345e-15, 88.191e-9, 1.004)	(0.9191e-15, 50.924e-9, 0.995)
PCI write	(17.737e-15, 40.461e-9, 0.969)	(7.2877e-15, 44.633e-9, 0.996)

Table 3. Coef£cients for the polynomial function to calculate the slowdown factor for combinations of external and internal, read and write operations, derived by the least-square-£tting method from Figure 2

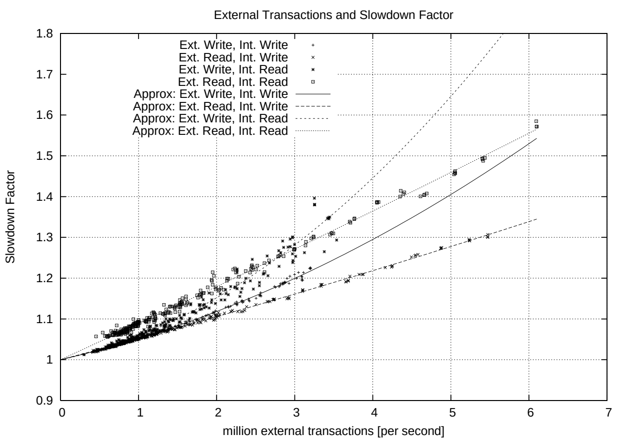


Figure 4. Approximation of resulting slowdown factors using polynomial functions and coef£cients from Table 3

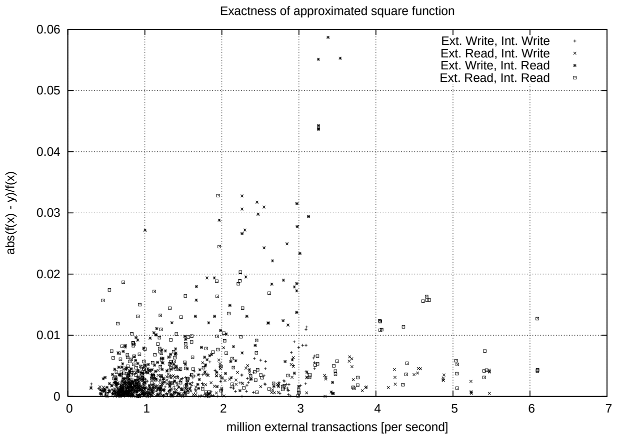


Figure 5. Exactness of the approximated square functions for the calculated slowdown factors

culated by

$$\sigma_y = \sqrt{\frac{1}{n-k-1}\sum_{i=1}^{n}\Big(y_i - f(x)\Big)^2}. \qquad (16)$$

The smaller the mean error, the better the approximation. In our case, we calculated $\sigma_{ww} = 0.003865$, $\sigma_{wr} = 0.013028$, $\sigma_{rr} = 0.007934$, and $\sigma_{rw} = 0.002696$. This also demonstrates the quality of our results.

Analyzing the graphs of Figure 2, we can also see that the impact on CPU read operations is different than on CPU write operations and also depends on the type (*i.e.*, read or write) of the external transaction. To further improve the method to calculate the application's WCSF presented in the previous section, we determine a *weighted slowdown factor* for an application based on the ratio of its read and write transactions. The ratio of external read transactions on the total number of transactions is

$$\rho_r = \frac{t_{ext}^r}{t_{ext}^r + t_{ext}^w}, \qquad (17)$$

and the ratio for write transactions is

$$\rho_w = \frac{t_{ext}^w}{t_{ext}^r + t_{ext}^w}. \qquad (18)$$

To calculate the weighted slowdown factor for internal read transactions, we add the slowdown factor for external read transactions ($\mathcal{F}_{rr}$) weighted by the read ratio (ρ_r) and the slowdown factor for external write transactions ($\mathcal{F}_{wr}$) weighted by the write ratio (ρ_r). We obtain the following:

$$\mathcal{F}_r(t_{ext}) = \mathcal{F}_{rr}(t_{ext}^r)\rho_r + \mathcal{F}_{wr}(t_{ext}^w)\rho_w \qquad (19)$$

for processor read operations. The same method is applied to obtain the weighted slowdown factor for write operations:

$$\mathcal{F}_w(t_{ext}) = \mathcal{F}_{ww}(t_{ext}^w)\rho_w + \mathcal{F}_{rw}(t_{ext}^r)\rho_r. \qquad (20)$$

We can determine the weighted slowdown factor of an application by replacing $\mathcal{F}_r^{L_{max}}$ and $\mathcal{F}_w^{L_{max}}$ in Equation 9 with $\mathcal{F}_r(t_{ext})$ and $\mathcal{F}_w(t_{ext})$, respectively. We receive:

$$\mathcal{F}_A(t_{ext}) = \frac{S_r cyc_r \mathcal{F}_r(t_{ext}) + S_w cyc_w \mathcal{F}_w(t_{ext}) + S_o cyc_o}{S_r cyc_r + S_w cyc_w + S_o cyc_o}. \qquad (21)$$

In the following section, we demonstrate how to apply these formulas to obtain the slowdown factor of an application under speci£c external load and compare it to measurement results.

4.3. An Example Calculation

Applying Equation 21 to the DES application with a very small worst-case slowdown factor gives a result almost equal to 1. This result can also be con£rmed by measurements which showed that DES is virtually unin¤uenced by any external load. Additionally, we performed a test of the memory-copy application with two PCI cards. While the £rst card generated 25MB/s PCI-bus read load ($t_{ext}^{r} = 1,562,500$ transactions per second), the second card generated 30MB/s PCI-bus write load ($t_{ext}^{w} = 937,500$ transactions per second). The memory-copy performance was about 23MB/s.[7] We measured the number of CPU read and CPU write transactions 6,029,312 for each transaction type.

We can calculate $\rho_r = 0.625$ and $\rho_w = 0.375$. Based on Equation 12, we can further calculate the individual slowdown factors as a combination of read and write operations by $\mathcal{F}_{ww} = 1.044$, $\mathcal{F}_{rw} = 1.081$, $\mathcal{F}_{wr} = 1.022$, and $\mathcal{F}_{rr} = 1.139$. Applying Equations 19 and 20 we obtain $\mathcal{F}_r = 1.095$ and $\mathcal{F}_w = 1.067$. Finally, the weighted slowdown factor is calculated by applying Equation 21: $\mathcal{F}_A(rd = 25\text{MB/s} + wr = 30\text{MB/s}) = 1.082$. This con£rms our measured slowdown factor of 1.08.

Instruction fetches from main memory are handled as read transactions, and therefore are automatically considered. If the size of the executed code is small enough to £t in the L2 cache, the contribution of instruction-fetch-based read transactions is very small. This is the typical case for most of the real-time application with a short path of periodically executed code. To prevent cache ¤ushing due to context switches, techniques such as cache coloring [11] can be used.

5. Related Work

We believe that this topic has a relevance for the real-time community, only two papers known to us describe the impact of input/output load (I/O-load) on the execution time of an application. However, both do not provide a general solution for dealing with this impact and do not exactly quantify it. The techniques described in Section 5.2 (Prediction of Execution Times) are used in Section 4 to derive an algorithmic approach to determine the impact of I/O load on the execution time of applications.

7 Remember that we still operate over uncached memory.

5.1. Impact on Application

Bellosa [4] discusses a simple approach for reducing the in¤uence of time-sharing applications on real-time applications on different processors in a multiprocessor system. If the frequency of non-bandwidth-bound memory accesses of a time-sharing application exceeds a certain threshold, the application is throttled by performing additional no-operation cycles. This reduces the load on the memory bus caused by this application, leading to a smaller impact on the applications executed on other processors.

An analytical approach to describe the impact of I/O load generated by VMEbus devices on real-time applications is described by Huang *et al.* [9]. They observed that all programs are composed of three basic structures: straight-line, conditions, and loops. Each block of a program is individually analyzed with regard to how its execution time is in¤uenced by I/O load. To bound the worst-case execution time of an entire program, the worst-case execution time of each basic block that may access I/O needs to be bounded. Since the bus controller uses a protocol based on the VMEbus speci£cation, which supports hard priorities to regulate the bus contention between the processor and the devices, the model cannot be used without modi£cations on commodity PC systems.

5.2. Prediction of Execution Times

To compare performance of software, processors, or whole computer systems, various techniques such as benchmark programs are available [2, 6, 1]. Since benchmark programs are designed to cover a wide range of application types, the results of one benchmark program are used to compare different machines. However, exact prediction of execution times of individual programs is not possible.

Several approaches for estimating and predicting the performance of applications on the same or other processors have been published [13, 12]. Saveedra and Smith [14] describe an approach by which the performance of an application is predicted based on *abstract operations*. The execution time of an application is the linear combination of the number of times each abstract operation is executed (C_i), multiplied by the time it takes to execute each operation (P_i). The *instruction mix* (C) gives the amount of each individual abstract operation. In an extension [15], they also consider effects of the memory hierarchy, such as caches and translation look-aside buffers (TLB) on the execution time. The execution time (T) of an application using an instruction mix with n individual abstract operation can be determined by:

$$T = \sum_{i=1}^{n} C_i P_i. \tag{22}$$

This methodology allows one to analyze the behavior and to characterize individual machines. Applications can be analyzed, and their execution time can be predicted on the characterized machines. Prediction accuracy is very good; often the difference between real and predicted execution time is less than 10%.

To characterize applications regarding their bus-load af£nity, we do not need to consider each individual operation. It is suf£cient to group instructions into memory and non-memory operations.

6. Summary & Future Work

In this paper we introduced the slowdown factor to express the impact of external bus load to the execution time of an application. The slowdown factor can be used to adjust a scheduler reservation to external load.

A quick, but coarse, estimation of whether an application can be scheduled on a system under any possible external load is based on the system's upper-bound worst-case slowdown factor. As is common for all worst-case estimations, the difference between actual and calculated (or estimated) value can be very high. An alternative uses the instruction mix, a description of an application, and the system's upper-bound worst-case slowdown factor. In contrast to the £rst method, the actual share of read and write instruction of an application is considered to determine an application-speci£c worst-case slowdown factor.

In the third method, the system's upper-bound worst-case slowdown factor is replaced by the worst-case slowdown factors for read and write operations. The fourth approach replaces these worst-case slowdown factors by the weighted slowdown factors under a certain load. The weighted slowdown factors are calculated by means of a polynomial approximation function. The result is an application-speci£c slowdown factor under a certain load.

On current PCI-based systems, the slowdown factors are small. However, with the advent of new high-speed bus systems the impact of external load on the execution time of applications will become relevant again. The described technique is a promising method to handle this impact and needs further investigation. Describing the memory behavior of an application based on its instruction mix has been veri£ed on modern processors. Extensions were made to cover unpredictabilities such as multi-level caches, translation look-aside buffers, branch predictions, and so on. It remains open how future architectures such as Intel's IA64 family with explicit parallel instruction coding (EPIC) and a variety of speculative execution techniques, or systems with hyper-threading (HT) can be described by this technique. Additionally, multiprocessor systems should be considered, where multiple processors can generate memory-bus load. In such systems, we expect a much higher impact on applications and a real relevance of this research.

7. Acknowledgment

I'd like to thank my thesis supervisor Hermann Härtig and the operating systems research group at Dresden University, Rich Uhlig and the systems research group at Intel MRL, Gil Neiger for proof-reading.

References

[1] The SPEC benchmark suite. `http://www.specbench.org`.

[2] AIM Multiuser Benchmarks. `http://www.caldera.com/developers/community/contrib/aim.html`. Suite VII and IX of what used to be AIM Technology's benchmarks, GPL-ed.

[3] R. Baumgartl, M. Borriss, H. Härtig, C.-J. Hamann, M. Hohmuth, L. Reuther, S. Schönberg, and J. Wolter. Dresden Realtime Operating System. In *Proceedings of the First Workshop on System Design Automation (SDA'98)*, pages 205–212, Dresden, Mar. 1998.

[4] F. Bellosa. Process cruise control: Throtteling memory access in a soft real-time environment. Technical Report TR-I4-97-02, IMMDV IV, University of Erlangen, July 1997.

[5] M. Benson. *PBI - PCI Bus Interface*. FORE Systems Inc., Nov. 1995.

[6] A. B. Brown and M. I. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–224, Seattle, WA, June 1997.

[7] S. Edgar and A. Burns. Statistical Analysis of WCET for Scheduling. In *Proceedings IEEE RTSS 2001*, London, UK, Dec. 2001. IEEE/IEE.

[8] H. Härtig, R. Baumgartl, M. Borriss, C.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, Sept. 1998.

[9] T.-Y. Huang, J. W. Liu, and J.-Y. Chung. Allow Cycle-Stealing Direct Memory Access I/O Concurrent with Hard-Real-Time Programs. In *Int. Conf on Parallel and Distributed Systems (ICSPAD)*, Tokyo, June 1996.

[10] Intel Corp., Santa Clara. *Intel 430TX PCISET: 82439TX System Controller (MTXC) Reference Manual*, Feb. 1997. Order #290559-001.

[11] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Third IEEE Real-time Technology and Applications Symposium (RTAS)*, pages 213–223, Montreal, Canada, June 1997.

[12] D. Ofelt and J. L. Hennessy. Ef£cient performance prediction for modern microprocessors. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ACM SIGMETRICS Performance Evaluation Review, pages 229–239, New York, June 2000. ACM Press.

[13] B. Peuto and L. Shustek. An instruction timing model of CPU performance. In *International Conference on Computer Architecture, 25 years of the international symposia*

on computer architecture (selected papers), pages 152–165, Barcelona, Spain, 1998. ACM Press, New York, NY, USA.

[14] R. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, Nov. 1996.

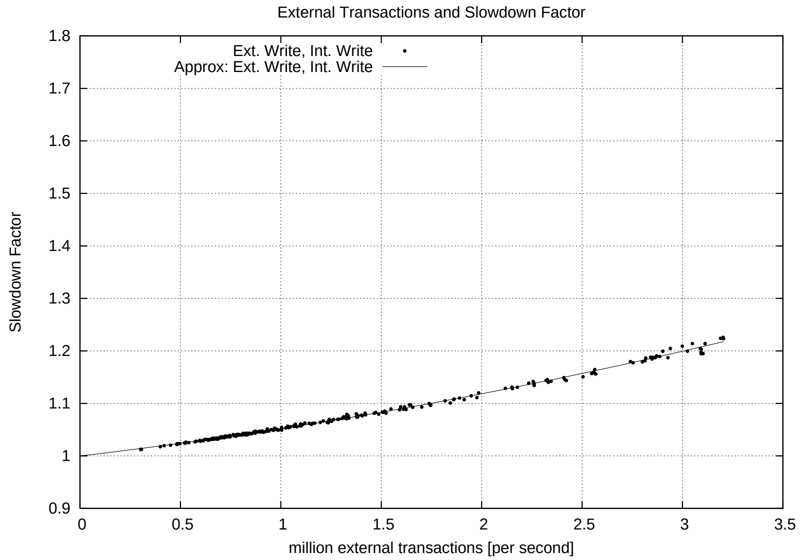
[15] R. H. Saavedra and A. J. Smith. Measuring cache and TLB performance and their effect on benchmark run times. *IEEE Trans. on Computers*, C-44(10):1223–1235, Oct. 1995.

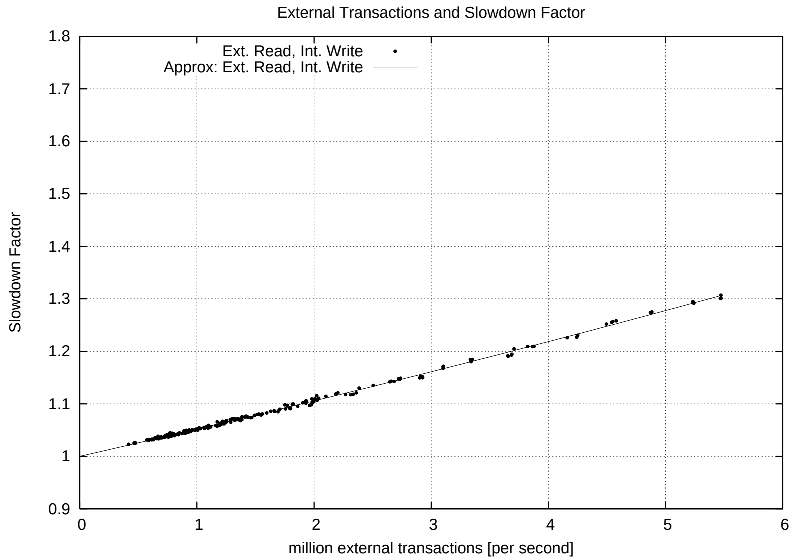
[16] S. Schönberg, F. Mehnert, C.-J. Hamann, L. Reuther, and H. Härtig. Performance and bus transfer in¤uences. In *First Workshop on PC-based System Performance and Analysis*, San Jose, CA, Oct. 1998. ACM.

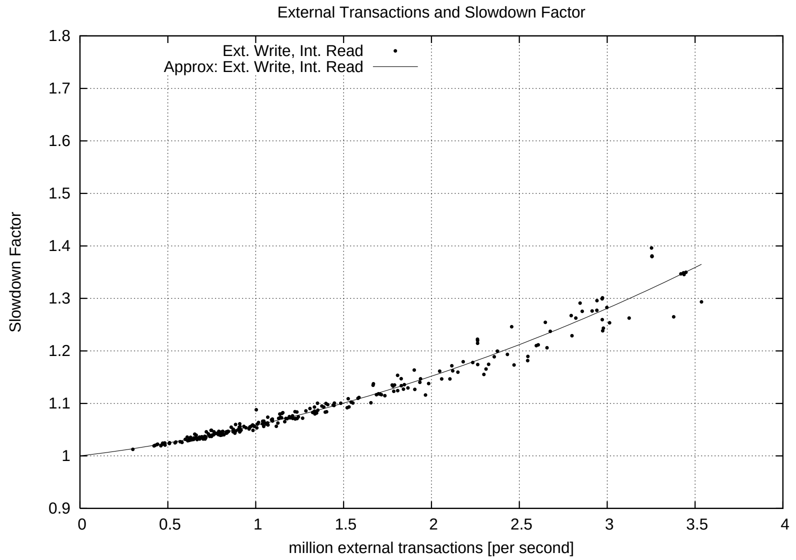
[17] R. Sedgewick. Implementing Quicksort programs. *Communications of the ACM*, 21(10):847–857, Oct. 1978. See corrigendum [18].

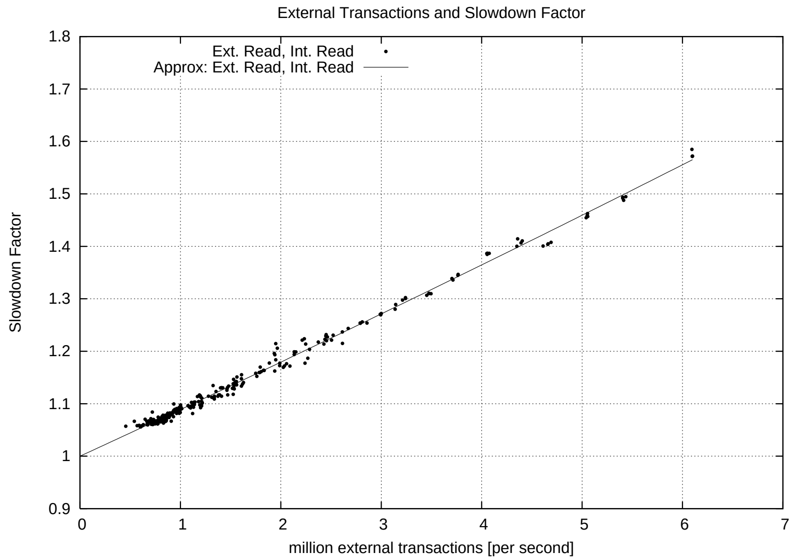
[18] R. Sedgewick. Corrigendum: "Implementing Quicksort Programs". *Communications of the ACM*, 22(6):368–368, June 1979. See [17].

[19] United States. National Bureau of Standards. *Data Encryption Standard*, volume 46 of *Federal Information Processing Standards publication*. U.S. National Bureau of Standards, Gaithersburg, MD, USA, 1977.


External Transactions and Slowdown Factor
Ext. Write, Int. Write
Approx: Ext. Write, Int. Write
Slowdown Factor
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0
0.5
1
1.5
2
2.5
3
3.5
million external transactions [per second]


External Transactions and Slowdown Factor
Ext. Read, Int. Write
Approx: Ext. Read, Int. Write
Slowdown Factor
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0
1
2
3
4
5
6
million external transactions [per second]


External Transactions and Slowdown Factor
Ext. Write, Int. Read
Approx: Ext. Write, Int. Read
Slowdown Factor
million external transactions [per second]
0.9
1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
0
0.5
1
1.5
2
2.5
3
3.5
4


External Transactions and Slowdown Factor
Ext. Read, Int. Read
Approx: Ext. Read, Int. Read
Slowdown Factor
million external transactions [per second]
0.9
1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
0
1
2
3
4
5
6
7