An Efficient Procedure for Building the Functional Performance Model of a Processor

Alexey Lastovetsky, Ravi Reddy, Robert Higgins

Department of Computer Science, University College Dublin, Belfield Dublin 4, Ireland
E-mail: Alexey.Lastovetsky@ucd.ie, Manumachu.Reddy@ucd.ie, Robert.Higgins@ucd.ie

Abstract---In this paper, we present an efficient procedure for building a piecewise linear function approximation of the speed function of a processor with hierarchical memory structure. The procedure tries to minimize the experimental time used for building the speed function approximation. We demonstrate the efficiency of our procedure by performing experiments with a matrix multiplication application and a Cholesky Factorization application that use memory hierarchy efficiently and a matrix multiplication application that uses memory hierarchy inefficiently on a local network of heterogeneous computers.

1. Introduction

In our previous research [1], we addressed the problem of optimal distribution or scheduling of computational tasks on networks of heterogeneous computers when one or more tasks do not fit into the main memory of the processors and when relative speeds of processors cannot be accurately approximated by constant functions of the problem size. We designed efficient algorithms to solve this scheduling problem using a performance model that integrates some of the essential features of a heterogeneous network of computers (HNOC) having a major impact on the performance, such as the processor heterogeneity, the heterogeneity of memory structure, and the effects of paging. Under this model, the speed of each processor is represented by a continuous and relatively smooth function of the problem size. This model is application-centric in the sense that generally speaking different applications will characterize the speed of the processor by different functions. Actually on general-purpose common heterogeneous networks,

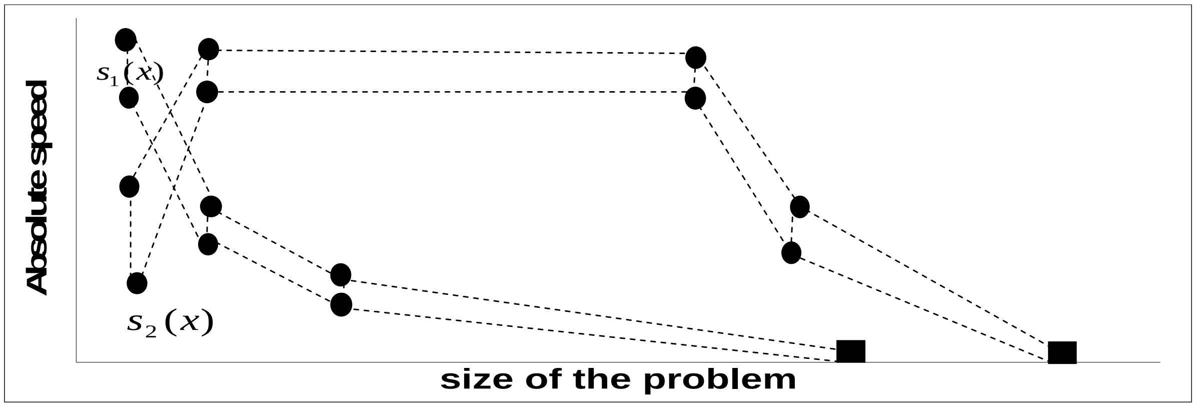


Figure 1. Using piecewise linear approximation to build speed bands for 2 processors. The circular points are experimentally obtained whereas the square points are calculated using heuristics. The speed band for processor $s_1(x)$ is built from 3 experimentally obtained points (application run on this processor uses memory hierarchy inefficiently) whereas the speed band $s_2(x)$ (application run on this processor uses memory hierarchy efficiently) is built from 4 experimentally obtained points.

an integrated computer will experience constant and stochastic fluctuations in the workload. This changing transient load will cause a fluctuation in the speed of the computer in the sense that the execution time of the same task of the same size will vary for different runs at different times. The natural way to represent the inherent fluctuations in the speed is to use a speed band rather than a speed function. The width of the band characterizes the level of fluctuation in the performance due to changes in load over time. In our previous research, we did not propose any methods to build and maintain the speed band of a processor.

In this paper, we present an efficient and a practical procedure for building a piecewise linear function approximation of the speed band of a processor with hierarchical memory structure. This band should be able to represent any speed function of the processor, that is, any speed function representing the performance of the processor should fit into the speed band. The procedure tries to minimize the experimental time used for building the piecewise linear function approximation of the speed band. We do not propose methods to maintain the speed function approximation. This is a subject of our future research.

Sample piece-wise linear function approximations of the speed band of a processor are shown in Figure 1. Each of the approximations is built using a set of few experimentally obtained points. The more points used to build the approximation, the more accurate the approximation is. However it is prohibitively expensive to use large number of points. Hence an optimal set of few points needs to be chosen to build an efficient piecewise linear function approximation of the speed band. Such an approximation gives the speed of the processor for any problem size with certain accuracy within the inherent deviation of the performance of computers typically observed in the network.

The rest of the paper is organized as follows. In section 2, we formulate the problem of building a piecewise linear function approximation of a processor and present an efficient and a practical procedure to solve the problem. To demonstrate the efficiency of our procedure, we perform experiments using a matrix multiplication application and a Cholesky Factorization application that use memory hierarchy efficiently and a matrix multiplication application that uses memory hierarchy inefficiently on a local network of heterogeneous computers.

2. Procedure for Building Speed Function Approximation

This section is organized as follows. We start with the formulation of the speed band approximation building problem. This is followed by a section on obtaining the load functions characterizing the level of fluctuation in load over time. The third section presents the assumptions adopted by our procedure. We then present some operations and relations related to the piecewise linear function approximation of the speed band. And finally we explain our procedure to build the piecewise linear function approximation.

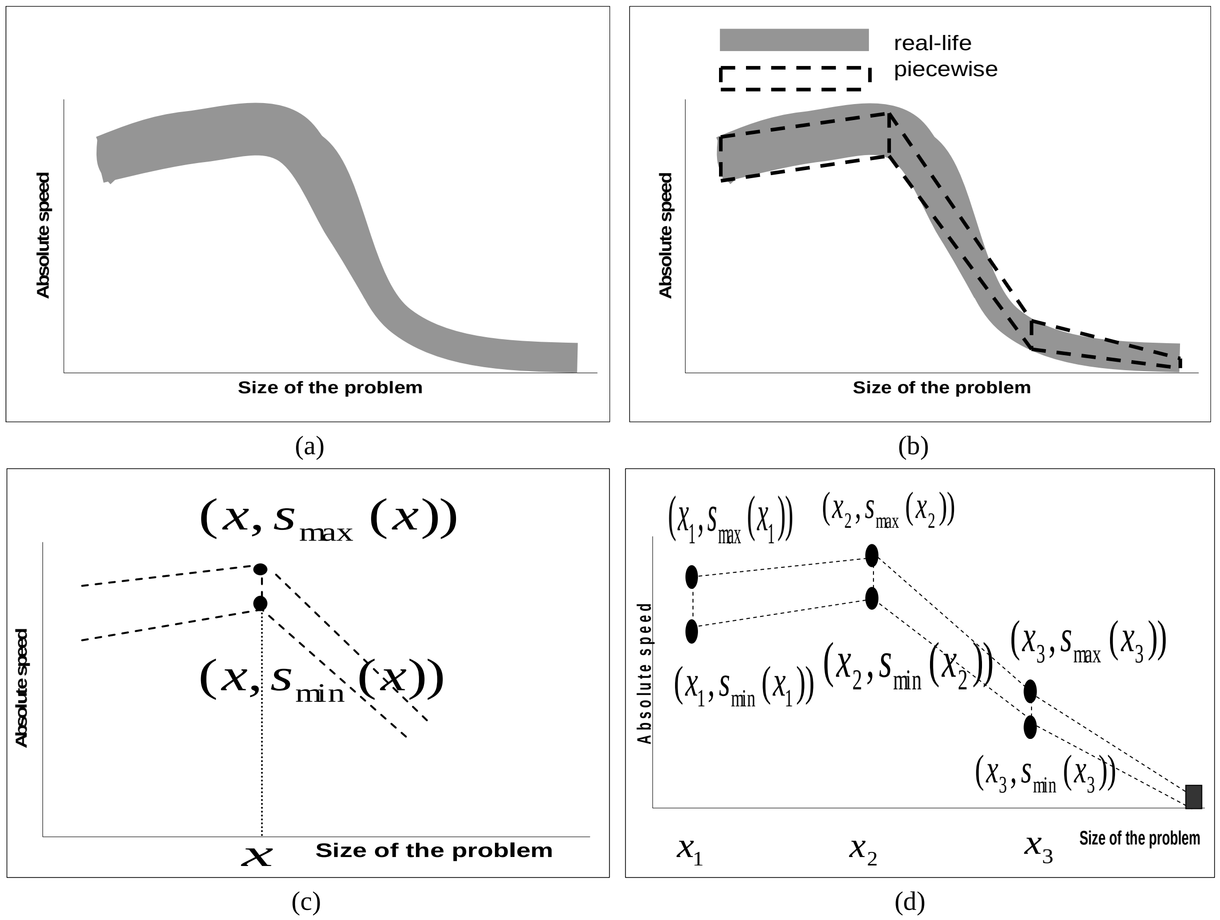


Figure 2. (a) Real-life speed band of a processor, (b) Real-life speed band of a processor and a piecewise linear function approximation of a processor, (c) The speeds $s_{max}(x)$ and $s_{min}(x)$ representing a cut of the real band used to build the piecewise linear approximation, and (d) Piecewise linear approximation built by connecting the cuts.

2.1 Problem Formulation

For a given application in a real-life situation, the performance demonstrated by the processor is characterized by a speed band representing the speed function of the processor with the width of the band characterizing the level of fluctuation in the speed due to changes in load over time. This is shown in Figure 2(a).

The problem is to find experimentally an approximation of the speed band of the processor that can represent the speed band with sufficient accuracy and at the same time spend minimum experimental time to build the approximation. One such approximation is a piecewise linear

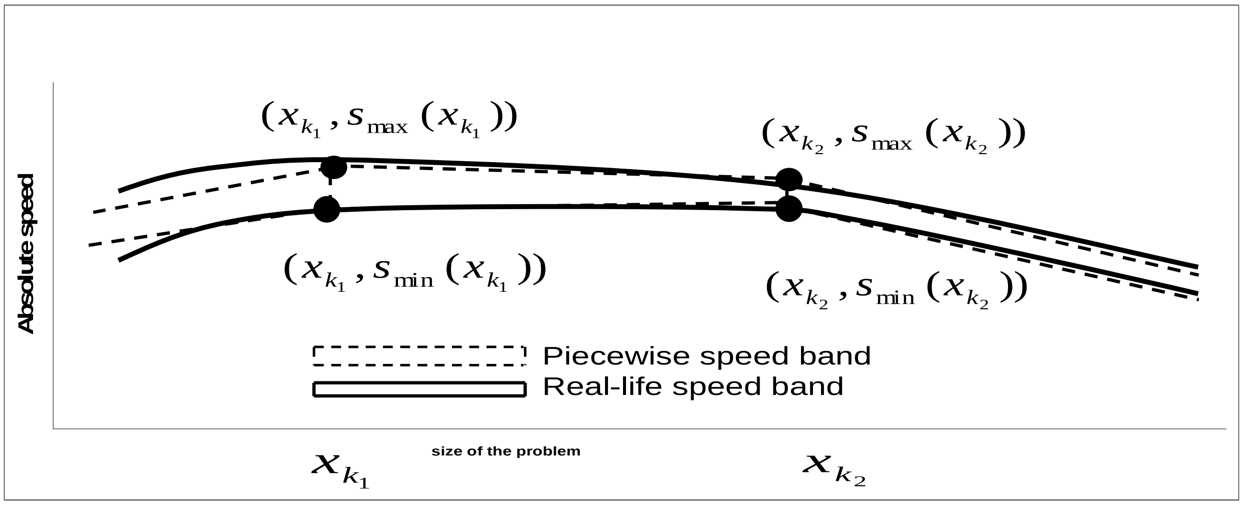

$(x_{k_1}, s_{\max}(x_{k_1}))$
$(x_{k_2}, s_{\max}(x_{k_2}))$
$(x_{k_1}, s_{\min}(x_{k_1}))$
$(x_{k_2}, s_{\min}(x_{k_2}))$
Absolute speed
Piecewise speed band
Real-life speed band
size of the problem
x_{k_1}
x_{k_2}

Figure 3. The non-empty intersectional area of piecewise linear function approximation with the real-life speed band is a simply connected surface.

function approximation which accurately represents the real-life speed band with a finite number of points. This is shown in Figure 2(b).

The piecewise linear function approximation of the speed band of the processor is built using a set of experimentally obtained points for different problem sizes.To obtain an experimental point for a problem size ***x*** (we define the size of the problem to be the amount of data stored and processed by the application), we execute the application for this problem size. We measure the ideal execution time $\boldsymbol{t_{ideal}}$ and not the real time of execution. We define $\boldsymbol{t_{ideal}}$ as the time it would require to solve the problem on a completely idle processor. For example on UNIX platforms, this information can be obtained by using the ***time*** utility or the ***getrusage()*** system call. The ideal speed of execution $\boldsymbol{s}_{ideal}$ is then equal to the volume of computations divided by $\boldsymbol{t_{ideal}}$. We assume we have the load functions of historical load data $\boldsymbol{l_{max}(t)}$ and $\boldsymbol{l_{min}(t)}$, which are the maximum and minimum load averages observed over increasing time periods. The load average is the number of active processes running on the processor at any time. We make a prediction of the maximum and minimum average load, $\boldsymbol{l_{max,predicted}}(\mathbf{x})$ and $\boldsymbol{l_{min,predicted}}(\mathbf{x})$ respectively, that would occur during the execution of the application for the problem size **x**. The creation of the functions ***l_{max}(t)*** and ***l_{min}(t)*** and predicting the load averages are explained in detail in the next section. Using ***s_{ideal}*** and the load averages predicted, we calculate ***s_{max}***(x) and ***s_{min}***(x) for a problem size ***x***:

$$S_{\max}(x) = S_{ideal}(x) - l_{\min,\,predicted}(x) \times S_{ideal}(x)$$
$$S_{\min}(x) = S_{ideal}(x) - l_{\max,\,predicted}(x) \times S_{ideal}(x)$$

The experimental point is then given by a vertical line connecting the points (**x**, **s_{max}**(x)) and (**x**, **s_{min}**(x)). We call this vertical line the "cut" of the real band. This is illustrated in Figure 2(c). The difference between the speeds **s_{max}**(x) and **s_{min}**(x) represents the level of fluctuation in the speed due to changes in load during the execution of the problem size **x**. The piecewise linear approximation is obtained by connecting these experimental points as shown in Figure 2(d). So the problem of building the piecewise linear function approximation is to find a set of such experimental points that can represent the speed band with sufficient accuracy and at the same time spend minimum experimental time to build the piecewise linear function approximation.

Mathematically the problem of building piecewise linear function approximation can be formulated as follows:

Definition 1. *Piecewise Linear Function Approximation Building Problem PLFABP(l_{min}(t),l_{max}(t)):* Given the functions ***l_{min}***(t) and ***l_{max}***(t) (***l_{min}***(t) and ***l_{max}***(t) are functions of time, characterizing the level of fluctuation in load), obtain a set of n experimental points representing the piecewise linear function approximation of the speed band of a processor, each point representing a cut given by (x_i,s_{max}(x_i)) and (x_i,s_{min}(x_i)) where x_i is the size of the problem and s_{max}(x_i) and s_{min}(x_i) are speeds calculated based on the functions ***l_{min}***(t) and ***l_{max}***(t) and ideal speed **s_{ideal}** at point **i**, such that:

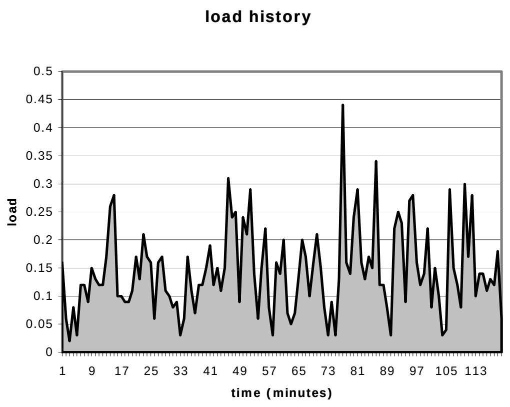
- The non-empty intersectional area of piecewise linear function approximation with the real-life speed band is a simply connected surface (A surface is said to be connected if a path can be drawn from every point contained within its boundaries to every other point. A topological space is simply connected if it is path connected and it has no holes. This is illustrated in Figure 3), and
- the sum $\sum_{i=1}^{n} t_i$ of the times is minimal where t_i is the experimental time used to obtain point **i**.

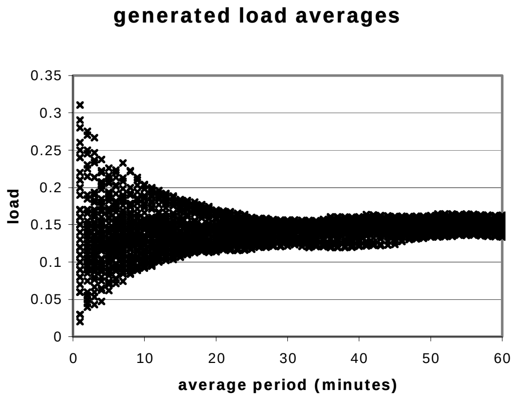
We provide an efficient and a practical procedure to build a piecewise linear function approximation of the speed function of a processor.

2.2 Load Functions

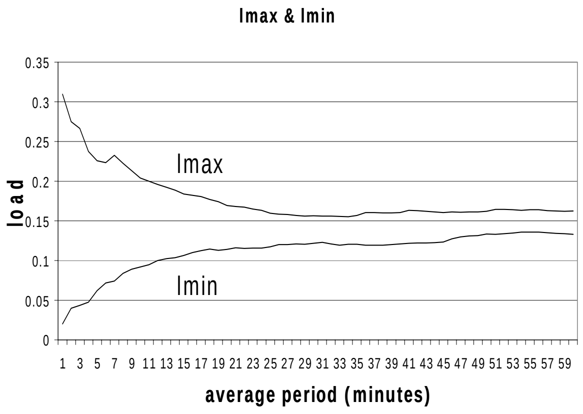
There are a number of experimental methods that can be used to obtain the functions $\boldsymbol{l_{min}(t)}$ and $\boldsymbol{l_{max}(t)}$ (characterizing the level of fluctuation in load over time) input to our procedure for building the piecewise linear function approximation.

One of the methods is to use the metric of Load Average. Load Average measures the number of active processes at any time. High load averages usually means that the system is being used heavily and the response time is correspondingly slow. The operating system maintains three figures for averages over one, five and fifteen minute periods. There are alternative metrics available through many utilities on various platforms such as **vmstat** (UNIX), **top** (UNIX), **perfmon** (Windows) or through performance probes and they may be combined to more accurately represent utilization of a system under a variety of conditions [2]. For this paper we will use the load average metric only.





(a) (b)



(c)

Figure 4. (a) *$l_{max}(t)$* and *$l_{min}(t)$* are generated from the load history. (b) A plot of points in matrix A. (c) *$l_{max}(t)$* and *$l_{min}(t)$*, the maximum and minimum loads calculated from the matrix of load averages A.

The load average data is represented by two piecewise linear functions: ***$l_{max}(t)$*** and ***$l_{min}(t)$***. The functions describe load averaged over increasing periods of time up to a limit ***w*** as shown in Figure 4(c). This limit should be at most the running time of the largest foreseeable problem, which is the problem size where the speed of the processor can be assumed to be zero (this is given by problem size ***b*** discussed in section 2.5). For execution of a problem with a running time greater than this limit, the values of the load functions at ***w*** may be extended to infinity. The functions are built from load averages observed every Δ time units. One, five or fifteen minutes are convenient values for Δ as statistics for these time periods are provided by the operating

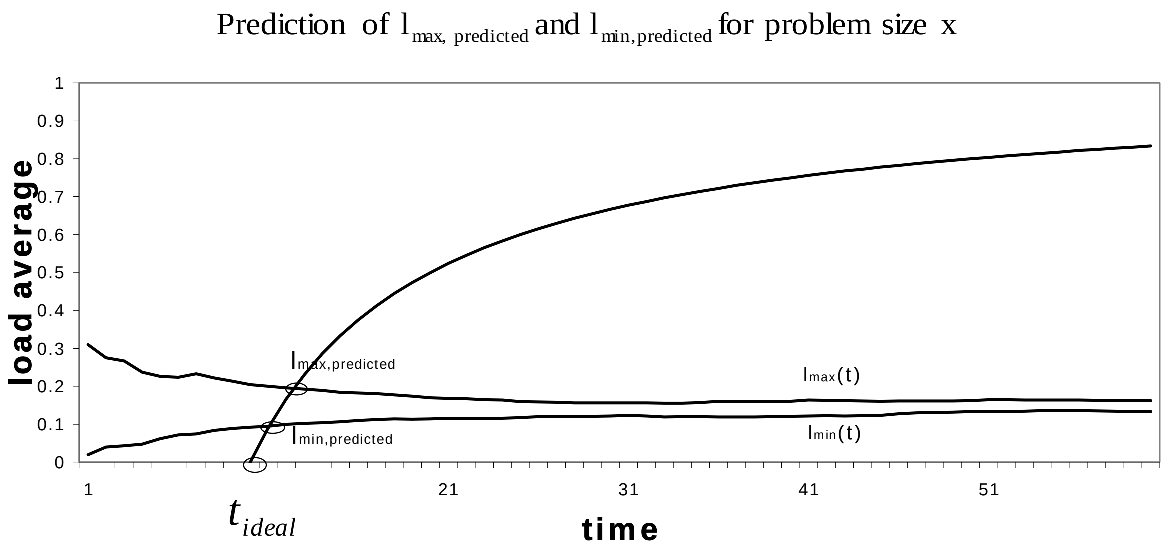


Figure 5. Intersection of load and running time functions (Formula 3).

system (using a system call ***getloadavg()***). Alternate values of Δ would require additional monitoring of the load average and translation into Δ time unit load average.

The amount of load observations used in the calculation of $\boldsymbol{l_{max}(t)}$ and $\boldsymbol{l_{min}(t)}$ is given by ***h***, the history. A sliding window with a length of ***w*** passes over the ***h*** most recent observations. At each position of the window a set of load averages is created from the observations inside the window. If Δ were one minute, a one minute average would be given by the first observation in the window, a two minute average would be the average of the first and second observations in the window, and so on. While the window is positioned completely within the history, a total of ***w*** load averages would be created in each set, the load averages having periods of $\Delta, 2\Delta, \ldots w\Delta$ time units. The window can move a total of ***w*** times, but after the ***(h – w)***-th time, its end will slide outside of the history. The sets of averages created at these positions will not range as far as $w\Delta$ but they are still useful. From all of these sets of averages, maximum and minimum load averages for each time period Δ, $\boldsymbol{2}\Delta$, … $\boldsymbol{w}\Delta$ are extracted and used to create the functions $\boldsymbol{l_{max}(t)}$ and $\boldsymbol{l_{min}(t)}$.

More formally, if we have a sequence of observed loads: $l_1, l_2, ..., l_h$, then the matrix $\boldsymbol{A}$ of load averages created from observations is defined as follows:

$$A = \begin{pmatrix} a_{1,1} & \cdot & \cdot & a_{1,h} \\ \cdot & & & \times \\ \cdot & & \times & \times \\ a_{w,1} & \times & \times & \times \end{pmatrix} \text{ where } a_{ij} = \frac{\sum_{k=j}^{i+j-1} l_k}{i \cdot \Delta}, \text{ for all } i = 1...h;\ j = 1...w \text{ and i + j < h} \qquad (1)$$

The elements marked as $\times$ in the matrix A are not evaluated as the calculations would operate on observations taken beyond l_{h}. $\boldsymbol{l_{max}(t)}$ and $\boldsymbol{l_{min}(t)}$, are then defined by the maximum and minimum calculated $\boldsymbol{j}$-th load averages respectively, i.e. the maximum or minimum value of a row $\boldsymbol{j}$ in the matrix (see Figure 4). Points are connected in sequence by straight-line segments to give a continuous piecewise function. The points are given by:

$$l_{\max}(j) = \max_{i=1}^{h}(a_{ij})$$
$$l_{\min}(j) = \min_{i=1}^{h}(a_{ij}) \qquad (2)$$

Initial generation of the array has been implemented with a complexity of $h \times (w)^2$. Maintaining the functions $\boldsymbol{l_{max}(t)}$ and $\boldsymbol{l_{min}(t)}$ after a new observation is made has a complexity of w^2. Δ, $\boldsymbol{h}$, and $\boldsymbol{w}$ may be adjusted to ensure the generation and maintenance of the functions is not an intensive task.

When building the speed functions $\boldsymbol{S_{min}(x)}$ and $\boldsymbol{S_{max}(x)}$, we execute the application for a problem size $\boldsymbol{x}$. We then measure the ideal time of execution $\boldsymbol{t_{ideal}}$. We define $\boldsymbol{t_{ideal}}$ as the time it would require to solve the problem on a completely idle processor. On UNIX platforms it is possible to measure the number of CPU seconds a process has used during the total time of its execution. This information is provided by the ***time*** utility or by the ***getrusage()*** system call. We assume that the number of CPU seconds a process has used is equivalent to time it would take to complete execution on a completely idle processor: t_{ideal}. We can then estimate the time of execution for the problem running under any load l with the following function:

$$t(l) = \frac{1}{1-l} \times t_{ideal} \qquad (3)$$

This formula assumes that the system is uniprocessor, that no jobs are scheduled if the load is one or greater and that the task we are scheduling is to run as a **nice**'d process (***nice*** is an operating system call that allows a process to change its priority), only using idle CPU cycles. These limitations fit the target of execution on non-dedicated platforms. If a job is introduced onto a system with a load of, for example, 0.1, the system has a 90% idle CPU, then the formula predicts that the job will take 1/0.9 times longer than the optimal time of execution: t_{ideal}.

In order to calculate the speed functions $S_{min}(x)$ and $S_{max}(x)$, we need to find the points where the function of performance degradation due to load (Formula 3) intersects with the history of maximal and minimal load $l_{max}(t)$ and $l_{min}(t)$ as shown in Figure 5. For a problem size **x**, the intersection points give the maximum and minimum predicted loads $l_{max,predicted}(\mathbf{x})$ and $l_{min,predicted}(\mathbf{x})$. Using these loads, the speeds $S_{min}(x)$ and $S_{max}(x)$ for a problem size x are calculated as:

$$S_{\max}(x) = S_{ideal}(x) - l_{\min,\,predicted}(x) \times S_{ideal}(x) \qquad (4)$$

$$S_{\min}(x) = S_{ideal}(x) - l_{\max,\,predicted}(x) \times S_{ideal}(x) \qquad (5)$$

where $S_{ideal}(x)$ is equal to the volume of computations involved in solving the problem size x divided by the ideal time of execution t_{ideal}.

2.3 Assumptions

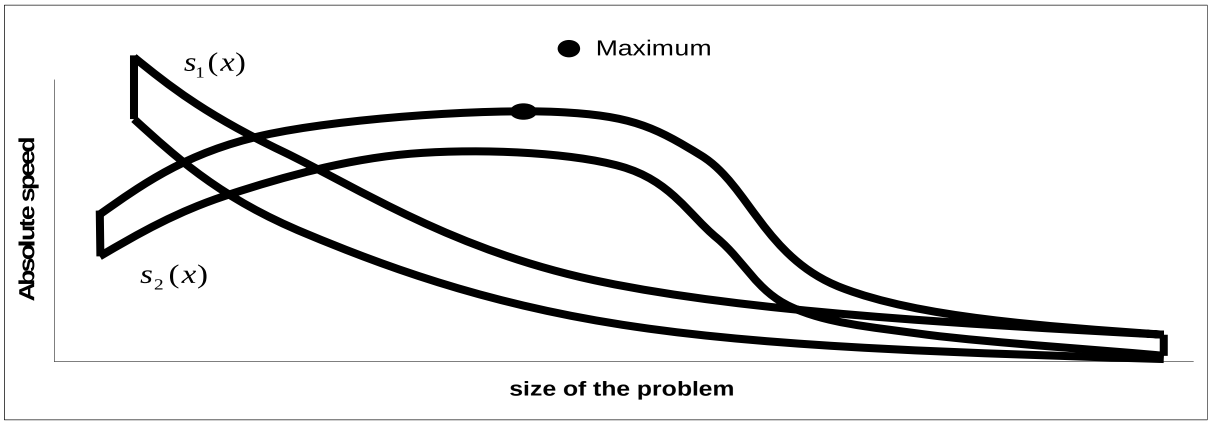
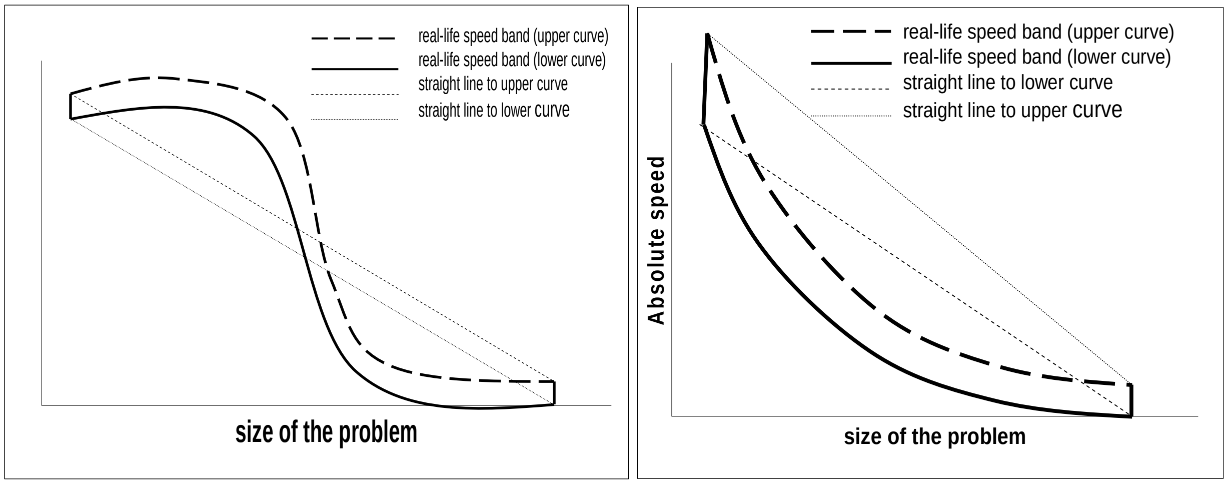


Figure 6. Permissible shapes of the graphs representing the real-life speed bands of two processors.



(a) (b)

Figure 7. (a) Shape of real-life speed function of processor for applications that use memory hierarchy efficiently, (b) Shape of real-life speed function of processor for applications that use memory hierarchy inefficiently.

We make some assumptions on the real-life speed band of a processor. Firstly, there are some shape requirements.

(a) We assume that the upper and lower curves of the speed band are continuous functions of the size of the problem.

(b) The permissible shapes of the real-life speed band are:

- The upper curve and the lower curve are both a non-increasing function of the size of the problem for all problem sizes (as shown by $s_1(x)$ in Figure 6).
- The upper curve and the lower curve are both a non-decreasing function of the size of the problem followed by a non-increasing function of the size of the problem (as shown by $s_2(x)$ in Figure 6).

(c) A straight line intersects the upper curve of the real-life speed band in no more than one point between its endpoints and the lower curve of the real-life speed band in no more than one point between its endpoints as shown for applications that use memory hierarchy efficiently in Figure 7(a) and for applications that use memory hierarchy inefficiently as shown in Figure 7(b).

(d) We assume that the width of the real-life speed band, representing the level of fluctuations in speed due to changes in load over time, decreases as the problem size increases.

These assumptions are justified by experiments conducted with a range of applications differently using memory hierarchy presented in [3].

Secondly, we do not take into account the effects on the performance of the processor caused by several users running heavy computational tasks simultaneously. We suppose only one user running heavy computational tasks and multiple users performing routine computations and communications, which are not heavy like email clients, browsers, audio applications, text editors etc.

2.4 Definitions

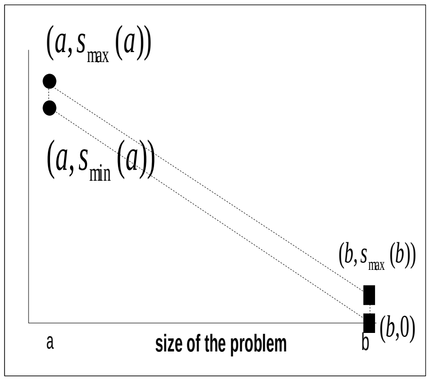
Before we present our procedure to build a piecewise linear function approximation of the speed band of a processor, we present some operations and relations on cuts that we use to describe the procedure. The piecewise linear function approximation of the speed band of the processor is built by connecting these cuts.

1. We use I_x at problem size x to represent the interval $(s_{min}(x),s_{max}(x))$. I_x is the projection of the cut C_x connecting the points $(x,s_{min}(x))$ and $(x,s_{max}(x))$ on the y-axis.
2. $I_x \le I_y$ if and only if $s_{max}(x) \le s_{max}(y)$ and $s_{min}(x) \le s_{min}(y)$.
3. $I_x \cap I_y$ represents intersection between the intervals $(s_{min}(x),s_{max}(x))$ and $(s_{min}(y),s_{max}(y))$. If $I_x \cap I_y = \emptyset$ where $\emptyset$ represents an empty set with no elements, then the intervals are disjoint. If $I_x \cap I_y = I_y$, then the interval $(s_{min}(x),s_{max}(x))$ contains the interval $(s_{min}(y),s_{max}(y))$, that is, $s_{max}(x) \ge s_{max}(y)$ and $s_{min}(x) \le s_{min}(y)$.
4. $I_x = I_y$ if and only if $I_x \le I_y$ and $I_y \le I_x$.

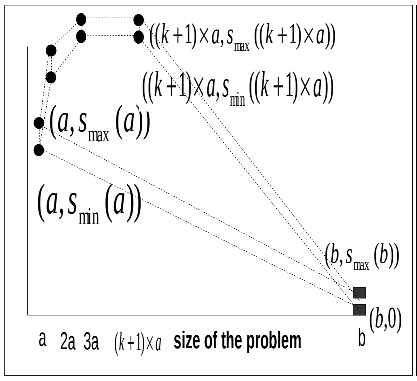
2.5 Speed Function Approximation Building Procedure

Procedure *Geometric Bisection Building Procedure GBBP($l_{max}(t)$, $l_{min}(t)$).* The procedure to build the piecewise linear function approximation of the speed band of a processor consists of the following steps and is illustrated in Figure 8:

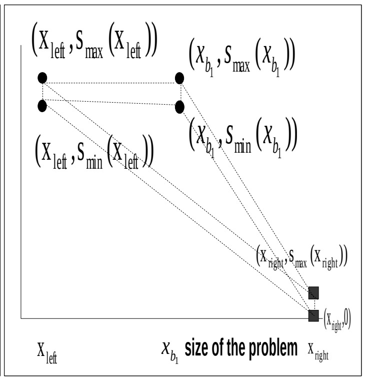
1. We select an interval [*a*,*b*] of problem sizes where *a* is some small size and *b* is the problem size large enough to make the speed of the processor practically zero. In most cases, *a* is the problem size that can fit into the top level of memory hierarchy of the computer (L1 cache) and *b* is the problem size that is obtained based on the maximum amount of memory that can be allocated on heap. To calculate the problem size *b*, we run a modified version of the application, which includes only the code that allocates memory on heap. For example consider a matrix-matrix multiplication application of two dense square matrices *A* and *B* of size **n**×**n** to calculate resulting matrix *C* of size **n**×**n**, the modified version of the application would just contain the allocation and de-allocation of



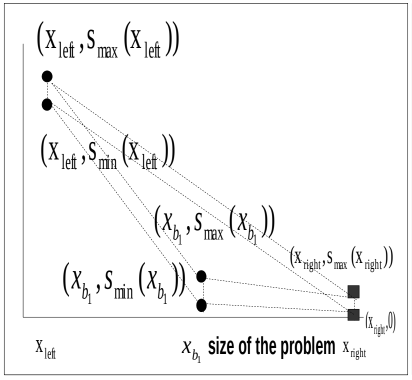
(a)


$((k+1)\times a, s_{\max}((k+1)\times a))$
$((k+1)\times a, s_{\min}((k+1)\times a))$
$(a, s_{\max}(a))$
$(a, s_{\min}(a))$
$(b, s_{\max}(b))$
$(b, 0)$
a 2a 3a $(k+1)\times a$ size of the problem
b

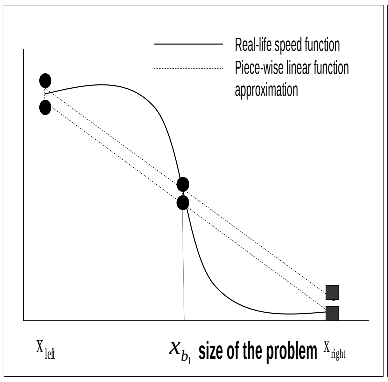
(b)


$(X_{\text{left}}, s_{\max}(X_{\text{left}}))$
$(x_{b_1}, s_{\max}(x_{b_1}))$
$(X_{\text{left}}, s_{\min}(X_{\text{left}}))$
$(x_{b_1}, s_{\min}(x_{b_1}))$
$(X_{\text{right}}, s_{\max}(X_{\text{right}}))$
$(X_{\text{right}}, 0)$
X_{left}
x_{b_1} size of the problem X_{right}

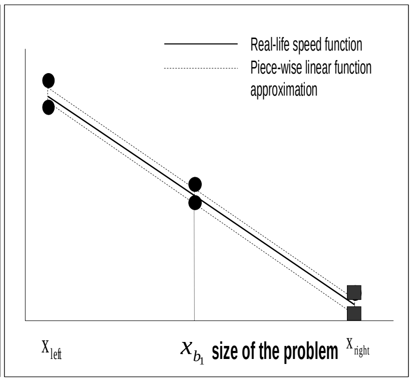
(c)


$(X_{\text{left}}, s_{\max}(X_{\text{left}}))$
$(X_{\text{left}}, s_{\min}(X_{\text{left}}))$
$(x_{b_1}, s_{\max}(x_{b_1}))$
$(X_{\text{right}}, s_{\max}(X_{\text{right}}))$
$(x_{b_1}, s_{\min}(x_{b_1}))$
$(X_{\text{right}}, 0)$
X_{left}
x_{b_1} size of the problem X_{right}

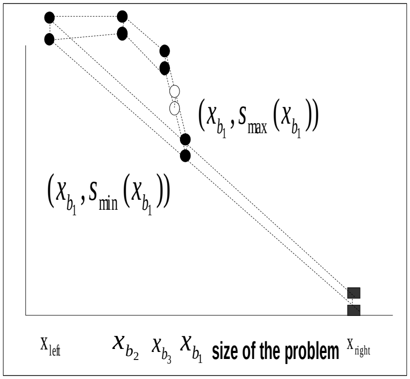
(d)


Real-life speed function
Piece-wise linear function approximation
X_{left}
x_{b_1} size of the problem X_{right}

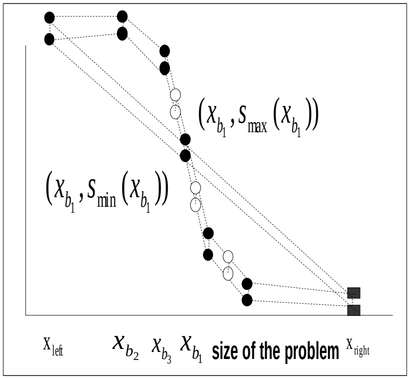
(e)


Real-life speed function
Piece-wise linear function approximation
X_{left}
x_{b_1} size of the problem X_{right}

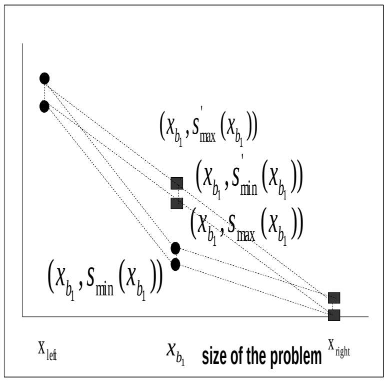
(f)


$(x_{b_1}, s_{\max}(x_{b_1}))$
$(x_{b_1}, s_{\min}(x_{b_1}))$
X_{left}
x_{b_2} x_{b_3} x_{b_1} size of the problem X_{right}

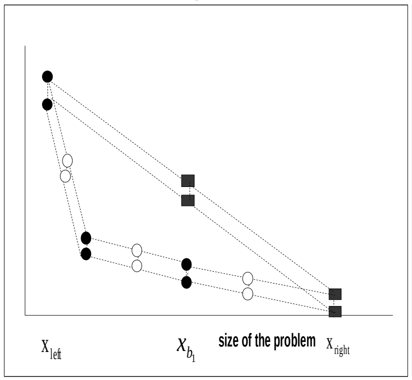
(g)


$(x_{b_1}, s_{\max}(x_{b_1}))$
$(x_{b_1}, s_{\min}(x_{b_1}))$
X_{left}
x_{b_2} x_{b_3} x_{b_1} size of the problem X_{right}

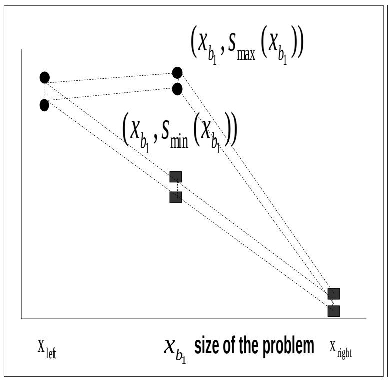
(h)


$(x_{b_1}, s'_{\max}(x_{b_1}))$
$(x_{b_1}, s'_{\min}(x_{b_1}))$
$(x_{b_1}, s_{\max}(x_{b_1}))$
$(x_{b_1}, s_{\min}(x_{b_1}))$
X_{left}
x_{b_1} size of the problem X_{right}

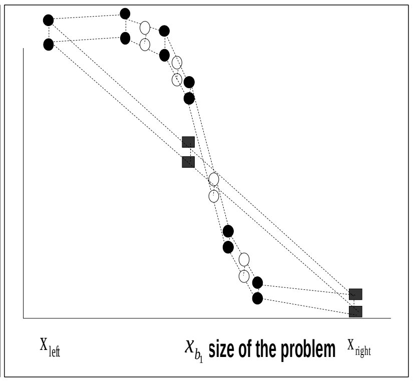
(i)



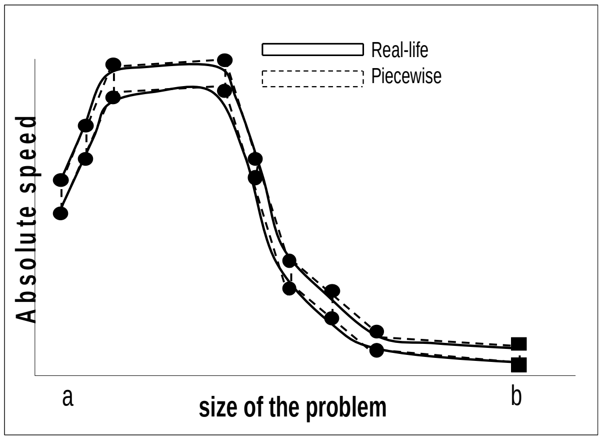
(j)


$(x_{b_1}, s_{\max}(x_{b_1}))$
$(x_{b_1}, s_{\min}(x_{b_1}))$
X_{left}
x_{b_1} size of the problem X_{right}

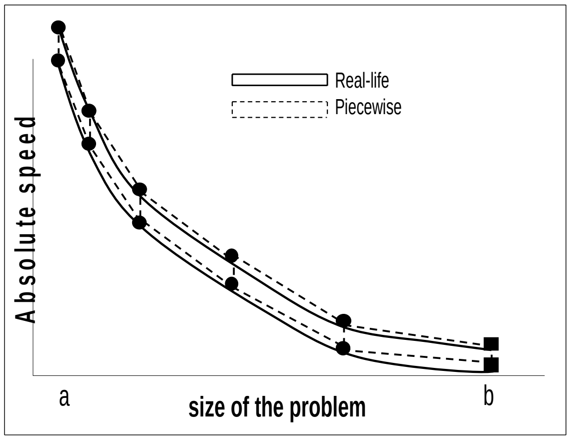
(k)



(l)



(m)



(n)

Figure 8. (a) to (l) Illustration of the procedure to obtain the piecewise linear function approximation of the speed band for a processor. Circular points are experimentally obtained points. Square points are points of intersection that are calculated but not experimentally obtained. White circular points are experimentally obtained and that fall in the current approximation of the speed band. (m) The final piecewise linear function approximation of the speed band of a processor for an application that utilizes memory hierarchy efficiently. (n) The final piecewise linear function approximation of the speed band of a processor for an application that utilizes memory hierarchy inefficiently.

matrices *A*, *B*, and *C* on heap. This modified version is then run until the application fails due to exhaustion of heap memory, the problem size at this point gives *b*. It should be noted that finding the problem size *b* by running the modified version should take just few seconds.

We obtain experimentally the speeds of the processor at point *a* given by $s_{max}(a)$ and $s_{min}(a)$ and we set the absolute speed of the processor at point *b* to 0. Our initial approximation of the speed band is a speed band connecting cuts C_a and C_b. This is illustrated in Figure 8(a).

2. We experiment with problem sizes *a* and 2*a*. If $I_{2a} \leq I_a$ or $I_{2a} \cap I_a = I_{2a}$, we replace the current approximation of the trapezoidal speed band with two trapezoidal connected bands, the

first one connecting the cuts C_a and C_{2a} and the second one connecting the cuts C_{2a} and C_b. We then consider the interval [2*a*,*b*] and apply step 3 of our procedure to this interval. The speed band in this interval connecting the cuts at problem sizes 2*a* and *b* is input to step 3 of the procedure. We set x_{left} to 2*a* and x_{right} to *b*.

If $I_a \leq I_{2a}$, we recursively apply this step until $I_{(k+1)\times a} \leq I_{ka}$ or $I_{(k+1)\times a} \leq I_{ka} = I_{(k+1)\times a}$. We replace the current approximation of the speed band in the interval [k×a,*b*] with two connected bands, the first one connecting the cuts C_{ka} and $C_{(k+1)\times a}$ and the second one connecting the cuts $C_{(k+1)\times a}$ and C_b. We then consider the interval [(k+1)×a,*b*] and apply the step 3 of our procedure to this interval. The speed band in this interval connecting the cuts $C_{(k+1)\times a}$ and C_b is input to step 3 of the procedure. We set x_{left} to (k+1)×a and x_{right} to *b*. This is illustrated in Figure 8(b).

It should be noted that the time taken to obtain the cuts at problem sizes {*a*, 2a, 3a,…,(k+1)×a} is relatively small (usually milliseconds to seconds) compared to that for larger problem sizes (usually minutes to hours).

3. We bisect this interval [x_{left},x_{right}] into sub-intervals [x_{left},x_{b_1}] and [x_{b_1},x_{right}] of equal length. We obtain experimentally the cut Cx_{b_1} at problem size x_{b_1}. We also calculate the cut of intersection of the line $x=x_{b_1}$ with the current approximation of the speed band connecting the cuts Cx_{left} and Cx_{right}. The cut of intersection is given by $C'x_{b_1}$.
    a. If $Ix_{left} \cap Ix_{b_1} \neq \emptyset$, we replace the current approximation of the speed band with two connected bands, the first one connecting the cuts Cx_{left} and Cx_{b_1} and the second one connecting the cuts Cx_{b_1} and Cx_{right}. This is illustrated in Figure 8(c). We stop building the approximation of the speed band in the interval [x_{left},x_{b_1}] and recursively apply step 3 for the interval [x_{b_1},x_{right}]. We set x_{left} to x_{b_1}.

b. If $Ix_{left} \cap Ix_{b_1} = \varnothing$ and $Ix_{right} \cap Ix_{b_1} \neq \varnothing$, we replace the current approximation of the speed band with two connected bands, the first one connecting the cuts Cx_{left} and Cx_{b_1} and the second one connecting the cuts Cx_{b_1} and Cx_{right}. This is illustrated in Figure 8(d). We stop building the approximation of the speed band in the interval $[x_{b_1}, x_{right}]$ and recursively apply step 3 for the interval $[x_{left}, x_{b_1}]$. We set x_{right} to x_{b_1}.

c. If $Ix_{left} \cap Ix_{b_1} = \varnothing$ and $Ix_{right} \cap Ix_{b_1} = \varnothing$ and $Ix_{b_1} \cap I'x_{b_1} \neq \varnothing$, then we have two scenarios illustrated in Figures 9(e) and 9(f) where experimental point at the first point of bisection falls in the current approximation of the speed band just by accident.

   Consider the interval $[x_{left}, x_{b_1}]$. This interval is bisected at the point x_{b_2}. We obtain experimentally the cut Cx_{b_2} at problem size x_{b_2}. We also calculate the cut of intersection $C'x_{b_2}$ of the line $x = x_{b_2}$ with the current approximation of the speed band. If $Ix_{b_2} \cap I'x_{b_2} \neq \varnothing$, we stop building the approximation of the speed function in the interval $[x_{left}, x_{b_1}]$ and we replace the current approximation of the trapezoidal speed band in the interval $[x_{left}, x_{b_1}]$ with two connected bands, the first one connecting the cuts Cx_{left} and Cx_{b_2} and the second one connecting the points Cx_{b_1} and Cx_{b_2}. Since we have obtained the cut at problem size x_{b_2} experimentally, we use it in our approximation. This is chosen as our final piece of our piece-wise linear function approximation in the interval $[x_{left}, x_{b_1}]$. If $Ix_{b_2} \cap I'x_{b_2} = \varnothing$, the intervals $[x_{left}, x_{b_2}]$ and $[x_{b_2}, x_{b_1}]$ are recursively bisected using step 3. Figure 8(g) illustrates the procedure.

Consider the interval [x_{b_1},x_{right}]. This interval is recursively bisected using step 3. We set x_{left} to x_{b_1}. Figure 8(h) illustrates the procedure.

d. If $Ix_{left} \cap Ix_{b_1} = \emptyset$ and $Ix_{right} \cap Ix_{b_1} = \emptyset$ and $Ix_{b_1} \leq I'x_{b_1}$ and $Ix_{b_1} \cap I'x_{b_1} = \emptyset$, we replace the current approximation of the speed band with two connected bands, the first one connecting the cuts Cx_{left} and Cx_{b_1} and the second one connecting the cuts Cx_{b_1} and Cx_{right}. This is illustrated in Figure 8(i). The intervals [x_{left},x_{b_1}] and [x_{b_1},x_{right}] are recursively bisected using step 3. Figure 8(j) illustrates the procedure.

e. If $Ix_{left} \cap Ix_{b_1} = \emptyset$ and $Ix_{right} \cap Ix_{b_1} = \emptyset$ and $I'x_{b_1} \leq Ix_{b_1}$ and $Ix_{b_1} \cap I'x_{b_1} = \emptyset$, we replace the current approximation of the speed band with two connected bands, the first one connecting the cuts Cx_{left} and Cx_{b_1} and the second one connecting the cuts Cx_{b_1} and Cx_{right}. This is illustrated in Figure 8(k). The interval [x_{left},x_{b_1}] and [x_{b_1},x_{right}] are recursively bisected using step 3. Figure 8(l) illustrates the procedure.

4. The stopping criterion of the procedure is satisfied when we don’t have any sub-interval to divide.

3. Experimental Results

We consider a Linux workstation and a Solaris workstation, which are integrated into local departmental network in the experiments. The specifications of the computer are shown in Table 1. The amount of memory, which is the difference between the total main memory and available main memory shown in the tables, is used by the operating system processes and few other user application processes that perform routine computations and communications such as email clients, browsers, text editors, audio applications etc. These processes use a constant percentage of CPU.

Table 1. Specifications of two processors.

Processor	Architecture	Cpu MHz	Total Main Memory (kBytes)	Available Main Memory (kBytes)	Cache (kBytes)	Matrix-matrix multiplication (dgemm) & Inefficient Matrix-matrix multiplication		Cholesky Factorization (dpotrf)	
						Size of matrix (n_a)	Size of matrix (n_b)	Size of matrix (n_a)	Size of matrix (n_b)
X1	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	1977	1033908	460368	512	100	13000	100	19500
X2	SunOS 5.9 UltraSPARC-Iii	440	524288	401408	2048	100	7000	100	13000

Table 2. Speedup of GBBP procedure over naïve procedure.

Processor	Matrix-matrix multiplication (ATLAS)	Cholesky Factorization (ATLAS)	Inefficient Matrix-matrix multiplication
	Speedup (Number of points taken to build using GBBP)	Speedup (Number of points taken to build using GBBP)	Speedup (Number of points taken to build using GBBP)
X1	8.5(7)	6.5(19)	5.9(5)
X2	5.7(10)	15(8)	5.7(5)

There are three applications used to demonstrate the efficiency of our procedure to build the piecewise linear function approximation of the speed band of a processor. The first application is Cholesky Factorization of a dense square matrix employing the LAPACK [4] routine **dpotrf**. The second application is matrix-matrix multiplication of two dense matrices using memory hierarchy inefficiently. The third application is based on matrix-matrix multiplication of two dense matrices employing the level-3 BLAS routine **dgemm** [5] supplied by Automatically Tuned Linear Algebra Software (ATLAS) [6].

Figures 9(a) to 9(f) show the real-life speed function and the piecewise linear function approximation of the speed band of the processors X1 and X2 for the matrix multiplication and Cholesky Factorization applications. The real-life speed function for a processor is built using a set of experimentally obtained points (**x**,**s**) . To obtain an experimental point for a problem size **x**, we execute the application for the problem size at that point. The absolute speed of the processor

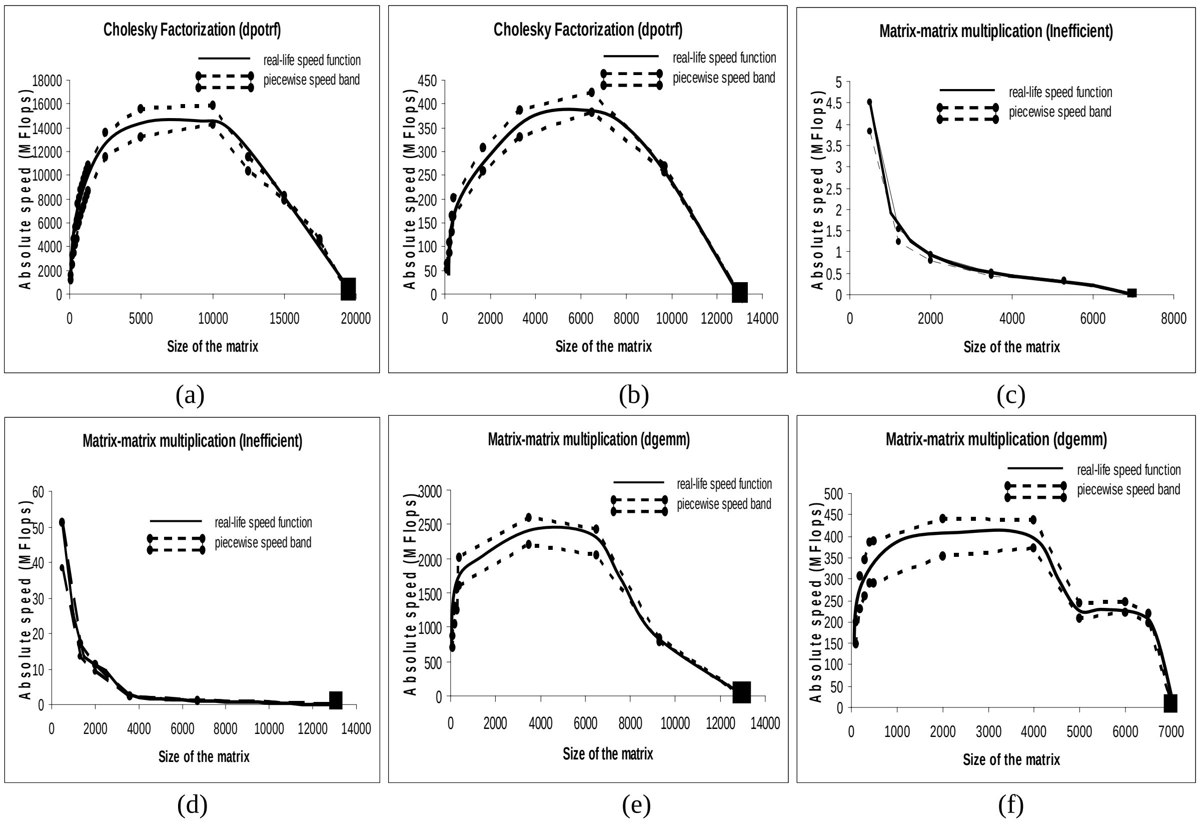


Figure 9. Piecewise linear approximation of the speed band against the real-life speed function. Circular points are experimentally obtained points. Square points are calculated but not experimentally obtained. (a) Cholesky Factorization using ATLAS on X1. (b) Cholesky Factorization using ATLAS on X2. (c) Matrix-matrix multiplication using memory hierarchy inefficiently on X1. (d) Matrix-matrix multiplication using memory hierarchy inefficiently on X2. (e) Matrix-matrix multiplication using ATLAS on X1. (f) Matrix-matrix multiplication using ATLAS on X2.

s for this problem size is obtained by dividing the total volume of computations by the real execution time (and not the ideal execution time).

Table 2 shows the speedup of Geometric Bisection Building Procedure (GBBP) over a naïve procedure. The naïve procedure divides the interval [*a*,*b*] of problem sizes equally into **n** points. The application is executed for each of the problem sizes {(a),(a+(b-a)/n),(a+2×(b-a)/n),…,(b)} to obtain the experimental points to build the piecewise linear function approximation of the

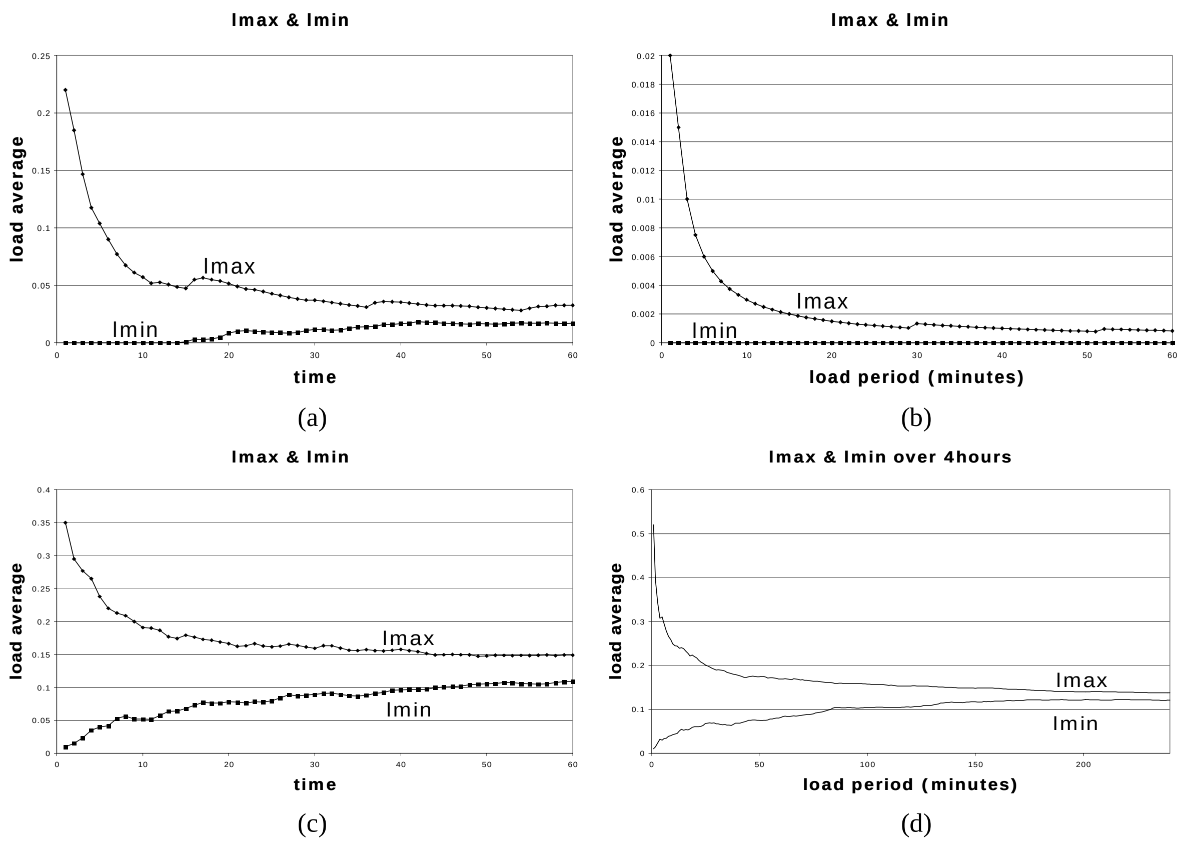


Figure 10. (a) Load functions for X1. (b) Load functions for X2. (c) Load functions for a departmental server running loads at all times. (d) Load functions generated with average periods beyond one hour.

speed band. In our experiments, we have used 20 points. The speedup calculated is equal to the ratio of the experimental time taken to build the piecewise linear function approximation of the speed band using the naïve procedure over the experimental time taken to build the piecewise linear function approximation of the speed band.

We measured the accuracy of the load average functions $\boldsymbol{l_{max}(t)}$ and $\boldsymbol{l_{min}(t)}$ by counting how often a future load was found to be within the bounds of the curves and by measuring the area between the curves. A very wide band will encompass almost all future loads but the prediction of maximum and minimum load will be poor. We fixed $\boldsymbol{w}$, the window size, and varied $\boldsymbol{h}$ to examine how the hit ratio and area of the band changed. X1, a machine operating as a desktop with constant minor fluctuations in load, shows that a 60 minute window size gives good accuracy with 4 hours of historical data. X2 is used for running intensive jobs with relative infrequency. Figures 10(a) and 10(b) shows a sample of the load functions for processors X1-X2. Figure 10(c) shows a load function for a departmental server with loads running at all times.

4. Discussion and Future Work

In this paper, we presented an efficient and a practical procedure to build a speed function approximation of a processor. We demonstrated the efficiency of our procedure by performing experiments with a matrix multiplication application and a Cholesky Factorization application that use memory hierarchy efficiently and a matrix multiplication application that uses memory hierarchy inefficiently on a local network of heterogeneous computers.

Most real-life speed bands shown by applications running on variety of operating systems satisfy the requirements of the GBBP procedure outlined in Section 2.3. However for some operating systems, the shape of the real-life speed band has a plateau in the region of paging as shown in Figure 9(f), which fails the requirement (c) of the GBBP procedure. This figure shows the real-life speed function and the piecewise linear function approximation of the speed band of an UltraSparc processor X2 for a matrix multiplication application using ATLAS. In this case, due to just one plateau in the region of paging, GBBP procedure manages to build piecewise linear function approximation. However it is inefficient since it takes two additional experimental points at problem sizes 5000 and 6500 in the region 4000-7000. In general, GBBP procedure fails to build an efficient piecewise linear function approximation for such shapes. We aim to extend our procedure to build piecewise linear function approximation efficiently for such shapes.

During the building of the piecewise linear function approximation using the GBBP procedure, we consider the cut of the real-life speed band experimentally obtained for a problem size is accurate enough if there is a non-empty intersectional area with cut of the current approximation of the speed band. That is if $I_x \cap I_y \neq \Phi$ where I_x and I_y represent the intervals $(s_{min}(x), s_{max}(x))$ and $(s_{min}(y), s_{max}(y))$ of reflections of cuts C_x and C_y on y-axis respectively. The procedure thus uses implicitly the notion of distance between the intervals to represent accuracy of the building procedure. This notion of distance between the intervals can be included in the parameter list to the GBBP procedure without any modifications to the procedure.

Further consideration should be put into choosing the maximum and minimum loads to represent a particular *n* minute load average. If the averages have a distribution that fits some curve then the maximum and minimum limits of the load functions could be set to encompass a certain percentage of this curve. This would result in a narrower pair of load curves and could give a more accurate representation of the band.

The general shape of $\boldsymbol{l_{max}(t)}$ and $\boldsymbol{l_{min}(t)}$ showed that for problems executing for very long time frames, beyond one hour to one day (shown in Figure 10(d)), the deviation between predicted maximum and minimum performance is very low. For example, at 4 hours the deviation is less than 1.6%. Variation in load average is very small at these time scales on all our machines, despite their differing roles. This would indicate that the importance of the band lies in scheduling jobs that run for lesser periods of time. The window size, ***w***, could be dynamically assigned so that the final pair of maximum and minimum load averages represents a variation in performance of some user-defined percentage, and after this point the band could be considered a constant function.

We understand the importance of the problem of efficient maintenance of the speed function approximation of the speed band. This problem is the subject of our current research. We aim to design efficient algorithms of data partitioning on heterogeneous networks of computers where the speed of a processor is represented by a speed band, the width of the band characterizing fluctuations in speed due to changes in load over time.

References

[1] A. Lastovetsky and R. Reddy, "Data Partitioning with a Realistic Performance Model of Networks of Heterogeneous Computers," In Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2004), 26-30 April 2004, New Mexico, France, CD-ROM/Abstracts Proceedings, IEEE Computer Society 2004.

[2] R. Wolski, N.T. Spring, and J. Spring, "Predicting the CPU Availability of Time-shared Unix Systems on the Computational Grid," In Cluster Computing, Volume 3, No. 4, pp.293-301, 2000.

[3] A. Lastovetsky, and J. Twamley, "Towards a Realistic Performance Model for Networks of Heterogeneous Computers," In International Symposium on High Performance Computational Science and Engineering HPSCE'04, Toulouse, France, August 22-27, 2004.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. LAPACK Users' Guide, Release 3.0, SIAM, Philadelphia, August 1999, ISBN:0-89871-447-8, Ref: http://www.netlib.org/lapack/lug.

[5] J. Dongarra, J. D. Croz, I. S. Duff, and S. Hammarling, "A set of level-3 basic linear algebra subprograms," In ACM Transactions on Mathematical Software, Volume 16, No. 1, pp.1-17, March 1990.

[6] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the atlas project," Technical report, Department of Computer Sciences, University of Tennessee, Knoxville, March 2000.